Design and Implementation of a new Inter-Process Communication Architecture for Log-based HIDS for 100 GbE Environments

---

Bachelor Thesis

by

Paul Raatschen

University of Potsdam
Institute for Computer Science
Operating Systems and Distributed Systems

Supervisors:
Prof. Dr. Bettina Schnor
M.Sc. Max Schrötter

Potsdam, April 12, 2023

Raatschen, Paul
raatschen@uni-potsdam.de
Design and Implementation of a new Inter-Process Communication Architecture for Log-based HIDS for 100 GbE Environments
Bachelor Thesis, Institute for Computer Science
University of Potsdam, April 2023

Thanks

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig angefertigt, nicht anderweitig zu Prüfungszwecken vorgelegt und keine anderen als die angegebenen Hilfsmittel verwendet habe. Sämtliche wissentlich verwendeten Textausschnitte, Zitate oder Inhalte anderer Verfasser wurden ausdrücklich als solche gekennzeichnet.

Potsdam, April 12, 2023

Paul Raatschen

Abstract

Deutsche Zusammenfassung

Deutsche Zusammenfassung

Contents

1 Introduction

Since the advent of the Internet, bandwiths available to both commercial and private users have been ever increasing. While this opens up new possibilitys for high bandwiths network applications, it also poses new security challenges for dealing with potentially malicious network traffic. In addition to traditional firewalls, Host-based Intrusion Detection System (HIDS) are a commonly used security measure, to protect a system . Traditionally, HIDS make use of application logfiles, which are parsed for information on possible attacks and to identify malicious clients. The Intrusion Prevention System (IPS) Fail2ban[1] is is an one of the most prim

The goal of this thesis will be the design and implementation of a new Inter-Process Communication (IPC) architecture for the transmission of log messages, that is able to facilitate low latency communication between sender and receiver. Additionally, the design should be able to scale to multiple recipients, in order to accommodate more complex security system, in which several processes require access to a hosts application log. For this purpose, a Proof of Concept IPS will be developed, that utilizes the proposed IPC architecture to receive log messages and ban malicious clients in the style of Fail2ban.

This thesis will be structured as follows: The following section provides background information on relevant concepts,

2 Background

The following section introduces the concept of HIDS with the specific example of Fail2ban and presents the problem setting this thesis aims to solve. In addition to that, an overview over common types of Inter-Process Communication and existing IPC based logging solution is given. Finally, external libraries and other software used for the implementation and evaluation of the proof of concept IPS are introduced.

2.1 Host-based Intrusion Detection / Prevention

Intrusion Detection Systems are tasked with monitoring and collecting data from target systems, which is then further processed and analyzed, to identify potential threads and facilitate a response [2]. The idea of specialized software for detecting intrusion attempts and other security threads goes as far back as 1980, when James Anderson published a study on "Computer security threat monitoring and surveillance", suggesting the use of automated tools to assist with security monitoring[3]. In 1987, Dorothy Denning presented a seminal model for Intrusion Detection Systems**IDS!** (**IDS!**), that proposed the use of pattern matching based on statistical analysis of audit records generated by a system, in order to detect abnormal user behavior [4]. Intrusion Detection Systems in general, can collect data from a multitude of sources. This allows for the distinction between network-based intrusion detection systems Network-based Intrusion Detection System (NIDS) and host-based introduction detection systems HIDS. NIDS monitor network interfaces and analyze captured traffic, while HIDS gather information directly provided by the hosts under their supervision. For the latter, this includes event logs of applications, as well as operating system Operating System (OS) based information, such as user logins, file system operations or systemcalls. For analysis of the accumulated data, there a two commonly deployed strategies: 1. Misuse based detection relies on predefined patterns of misuse or malicious behavior, which are then matched against the observed data. 2. Anomaly based detection uses statistical analysis to identify significant deviations from normally observed behavior, which, in principal, enables the detection of attack patterns, that have not been previously observed [2].

2.1.1 Fail2ban

Fail2ban is a open source Intrusion Prevention System IPS, that is widely used to protect hosts against a range of network-based attacks, such as, for instance, brute-force login attempts[1]. Intrusion prevention system constitute a special class of **IDS!**, that not only detects an intrusion attempt, but also initiates an active response, with the aim of preventing or mitigating the attack. To identify potentially malicious clients, Fail2ban uses a misuse detection approach based on application logs. For configured applications, Fail2ban actively monitors their log and parses new entries based on a predefined filter. Fail2ban uses configuration units called 'jails' , that allow for the customization to different applications. A jail defines a path to a application log,

the filter being applied to the log messages within the logfile and an action, that is executed on client matching the filter criteria. In addition to that, jails contain further parameters, such as the threshold of matches a client need to reach, in order for the action to be executed, as well as the duration of that action. The filter component of a jail defines a set of regular expressions Regular Expression (REGEX), that are used to identify certain events in a log, for example an unsuccessful login attempts or the exceeding of a request rate limit. The filter also obtains a clients IP address, as well as the date and time of the log messages, to determine, if the event occurred within a relevant time frame. Most commonly, the action issued by Fail2ban for eligible clients, is to ban the clients IP address for the duration configured in the associated jail. Fail2ban facilitates this via an iptables entry. Iptables is a utility program for Linux systems, that allows the interaction with the netfilter Kernel framework to implement packet filtering rules. When banning a client, Fail2ban add a iptables rule for the clients IP address that leads to the dropping of all subsequent network packets from that address, for as long as the rule is active.

Fail2bans netfilter based approach of packet filtering has disadvantage of scaling poorly for large traffic rates, since packets need to traverse several processing steps of the kernel network stack before they are ultimately discarded. In his master thesis, Florian Mikolajczak therefore proposed the usage of eBPF programs, as a more efficient way of implementing packet filtering for intrusion prevention systems[5]. The extended Berkley Packet Filter extended Berkeley Packet Filter (eBPF) is an interface of the Linux kernel, that allows the event based execution of verified user defined programs within the kernel. Packet filtering through eBPF can be facilitated with an eBPF program, that is executed on incoming packets and delivers a verdict, of wether the packets should be dropped or further processed by the Kernel. Via the Express Datapath eXpress Data Path (XDP), eBPF programs can be attached to different hooks in the packet processing pipeline. For supported devices, programs can be attached in XPD_DRIVER mode, where they are executed as part of the network device diver routine, very early into the packet processing. Florian Mikolajczak adapted an existing eBPF program by (name []) to handle IP version 6 Internet Protocol Version 6 (IPv6) as well and integrated it with Fail2ban to replace netfilter-based packet filtering. While he was able to demonstrate significant performance improvements over netfilter-based filtering, his measurents revealed, that Fail2ban has significant performance issues, when faced with a large influx of log messages.

Figure **??** shows the results of the Fail2ban measurement conducted by Florian Mikolajczak. In the experiment, a BIND9 DNS Server received unwanted requests at a rate of 1 million packets per second Packets per Second (PPS) from 254 different clients, which resulted in corresponding entries in the rate-limit log. Fail2ban was configured to ban clients with rate limit violations for 300 seconds. Initially, the performance was as expected. However, after the end of the first ban cycle, Fail2ban failed to renew the ban for some clients, leading to a significant amount of unwanted traffic still reaching the application. Closer inspection of Fail2bans behavior indicated, that the performance issues are a result of slow logfile parsing. Fail2bans rate of processing log entries appeared to be exceeded by rate of new entries, leading to Fail2ban falling increasingly further behind in the processing of logged events. This constitutes a problem, as it essentially makes Fail2ban and by extension the protected host, vulnerable to **DOS!** (**DOS!**) attacks. The primary goal of this thesis, will therefore be the development of an IPC-based logging architecture, that is able to handle the scenario above.

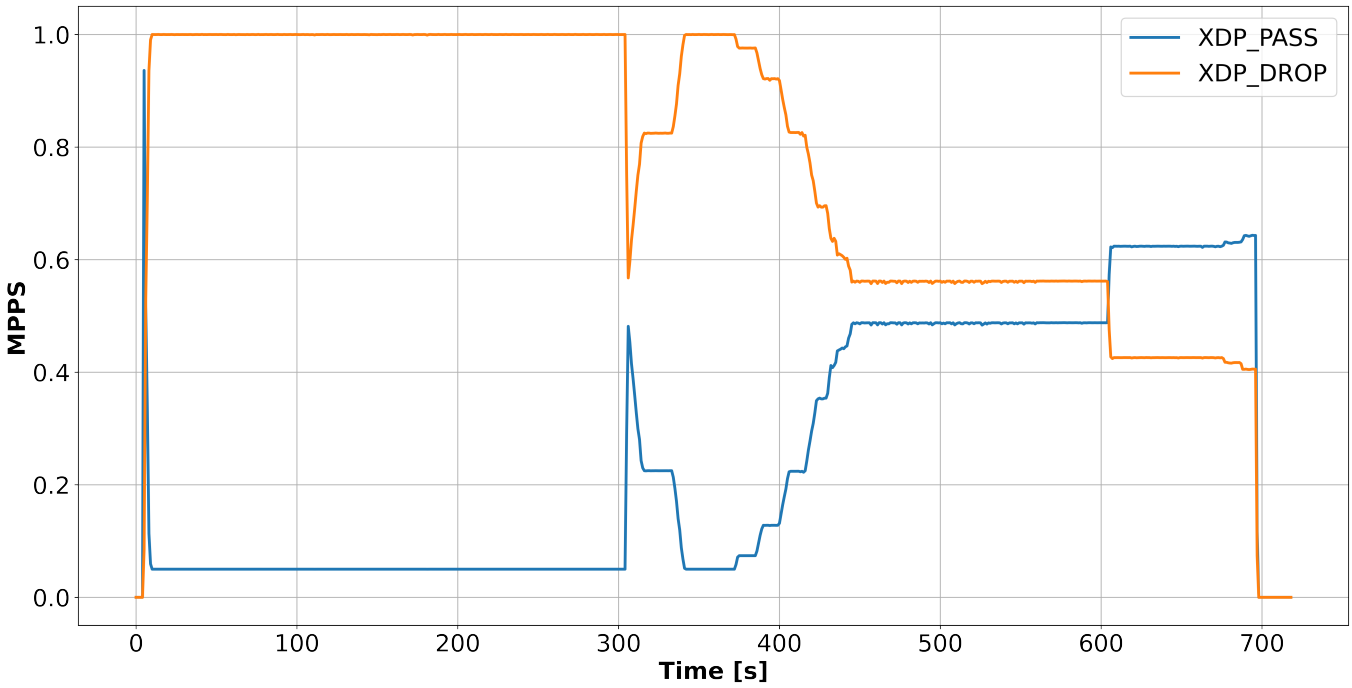


Figure 2.1: Results of experiment 1 from the master thesis of Florian Mikolajczak[5]. Fail2Ban, even in conjunction with more efficient eBPF filtering, performs poorly for large traffic rates.

2.2 Inter-Process Communication

2.2.1 Types of IPC

Inter-process communication allows the exchange of data between different processes through APIs provided by the operating system. Since the the development environment for this thesis will be Linux, I will focus on IPC APIs that are available on UNIX-like systems. A commonly used mechanism for sharing large amount of data between two processes on the same system is shared memory [6, p.301ff.]. Shared memory allows the allocation of memory segment in RAM, that can mapped into address space of several processes. The memory segment has associated file associated permissions, that allow the creating process to control read and write access by other processes. Linux provides two main ways of creating shared memory in the System V shared memory API [7] and the newer POSIX shared memory API [8]. Processes can essentially treat shared memory like any other valid memory in their address space, which allows for flexible application. A disadvantage of shared memory, at least for the aforementioned APIs, is its restriction to the local system boundaries, though solutions for network based remote direct memory access (RDMA) exist[9].

Pipes, specifically named pipes, are a way of facilitating unidirectional data transfer between processes on a UNIX system. Named pipes have a read and a write end and are associated with a special file in the filesystem, which can be accessed by different processes. Pipe can be written to and read from via the write and read system calls. The data is transmitted in a first-in first-out (FIFO) manner, hence why named pipes are also referred to as FIFOs. The maximum capacity up

to which a pipe can buffer data is limited, which by default is 65535 bytes on linux. [10]

Sockets are another common**ICP!** (**ICP!**) type for data transfer, that allow for one-to-one as well as one-to-many communication via a range of protocols[11, p.57ff.]. Sockets send data in formatted packets, that can be transmitted on a local host or via a network. For communication on a local system, UNIX systems offer UNIX Domain sockets. UNIX Domain sockets can be bound to a valid path in the filesystem, which serves the way of addressing the socket from another processes [12]. For data transfer beyond the local system, the Internet protocol Internet Protocol (IP) in conjunction with the Transmission Control Protocol Transmission Control Protocol (TCP) or User Datagram Protocol User Datagram Protocol (UDP) is most common. TCP offers connection oriented data transmission, that requires an orderly connection establishment through a handshake between the commutating parties. It further ensures a reliable transfer of data, by detecting transmission errors trough acknowledgment and retransmitting unacknowledged packets. UDP in contrast, provides no guarantees on reliable transfer, with the benefit of lower latency and less communication overhead. [11, p.29ff.]

Finally, message queues are another commonly used IPC mechanism that allow the exchange of messages between processes. Linux natively supports the System V [7] and POSIX [13] message queue APIs, but third party implementations exist as well, for instance the ZeroMQ message queue library [14]. Message queues essentially function as a buffer, which can store messages up to a certain capacity. A writing process adds messages to the queue, while reading processes remove messages from the queue. Unlike with sockets, writing and reading can occur asynchronously i.e. messages do not have to be read immediately after being added to a queue. ZeroMQ specifically supports a range of communication patterns, among them the publish-subscribe pattern, where a writer sends messages to multiple subscribed readers or the request-reply pattern, that can be used to implement remote procedure calls.

2.2.2 IPC based logging

Other than traditional logfiles,
Syslog, Rsyslog

2.3 Special Software

2.3.1 Hyperscan

Hyperscan is a open source regular expressions matching engine developed by Intel. It is specifically designed for high performance use cases, such as the application in security contexts and is being used by the intrusion detection systems Snort and Suricata [15]. The process of regular expressions matching with Hyperscan is separated into compile- and run-time. At compile-time a set regular expressions in string representation are compiled into a database, with additional configuration options

2.3.2 io_uring

2.3.3 Trex

3 Design & Implementation

The following section introduces the design and implementation of the proposed IPC architecture and the proof of concept IPS, as well as auxiliary libraries and applications. First, the requirements and overall purpose of the design are defined and translated into an abstract architecture. Subsequently, the implementation for a concrete IPC type is presented.

3.1 Requirements

The primary goal of the new IPC based logging architecture is the ability to handle high load scenarios, like the Denial of Service (DoS) scenario discussed in section 2.1.1. This includes the basic functional requirement of offering an API that can be used to send data to and receive data from different processes. Beyond that, the architecture should be usable in a realistic context. For this purpose, the following requirements should be satisfied, in order of importance.

- **Low latency**
  As discussed in 2.1.1, the primary issue of file based logging in the context of IPS appears to be a lack of transmission speed. Hence, offering low latency data transfer is a critical aspect of the new architecture. In a DoS scenario, the IPS need to receive information about malicious clients as fast as possible, in order to take countermeasures that lower the load on the system.

- **Low overhead**
  Another problem with file based logging is, that both read and write operations require systemcalls, which lead to costly context switches between application and kernel. Using traditional file I/O, read and write operations are blocking, meaning that a program can only resume, once the operation has concluded. In a high load scenario, where potentially millions of log messages are written per second, this can cause significant communication overhead for both reader and writer. It also looses critical execution time to waiting, which, on the side of the writing server, lowers the availability of the service and on the reading IPS side, reduces responsiveness to the ongoing attack. The new architecture should there offer low overhead communication, ensuring that both writer and reader are not spending a majority of their execution time writing or receiving log messages.

- **High bandwidth**
  In conjunction with low latency, high bandwidth is required to ensure a fast transfer of large amounts of data. When processing millions of log events per second, the architecture needs to be able to execute a single data transfer quickly, but also allow for a large amount of data to be transmitted at once, to avoid a bottleneck between writer and reader.

- **Reliability**
  While not the primary concern, the architecture should in principal be able to transfer log messages reliably, even under a high load. Hence, message loss or corruption should be kept to a minimum. This is to ensure, that the IPS or other reading application are aware of entirety of all logged events and no potentially security relevant event is missed.

- **Scalability**
  An advantage of file based logging is, that logged events are stored persistently and also accessible to a multitude of applications. Log data generated by an application may be relevant in different contexts, such as intrusion prevention, Security Information and Event Management (SIEM) or other non-security related uses. Therefore, the architecture should be able to scale to multiple readers that can access the log data in parallel.

- **Integrability / Portability**
  Finally, to be of use in a realistic context, it is required that the architecture is able to integrate with existing applications. Since the goal if this thesis is a proof of concept and not an API ready for use in production, this will be a lower priority requirement. However, the design should at a minimum offer a well defined and easily usable API, that can be realistically integrated into a real application and potentially further developed beyond the scope of this thesis. As part of that, the design should be portable to different systems and ideally not rely on special hardware of software support, other than what is commonly present on most Linux systems. The architecture should also be flexible i.e. impose as little restrictions as possible on the user, in order to be adaptable to a wide range of use case scenarios.

The architecture proposed on this thesis will likely not be able to satisfy all listed requirements to the fullest. Potential tradeoffs between requirements exists and will have to be considered in the following design. Where tradeoffs are impossible to avoid, the architecture should ideally offer configuration option, that allows the user to decide, which requirements should be prioritized.

3.2 Abstract Architecture

To further illustrate the purpose and requirements of the desired architecture, an abstract architecture is provided. The goal is to provide a high-level design, that formalizes a set of functional requirement, but leaves enough flexibility to be implemented with different IPC solutions.

Figure **??** presents the abstract design for the proposed IPC architecture. The primary assumed use case is a producer consumer scenario, in which a single application writes log messages that are read by k receiving processes. The transfer of data is assumed to be unidirectional, i.e. the roles of reader and writer are static for the duration of the communication. An exception to this can be auxiliary messages for the setup, teardown or reconfiguration of the communication, that may require readers to send data to the writer. The writer is assumed to be a server, that receives receives client request over a network. On modern multi-core system, web servers like nginx and apache or a DNS server such as BIND usually use multiple threads to handle incoming request. This allows for the parallel distribution of work onto the available CPU cores, making more efficient use of the systems resources. To efficiently accommodate a multi-threaded writer, the

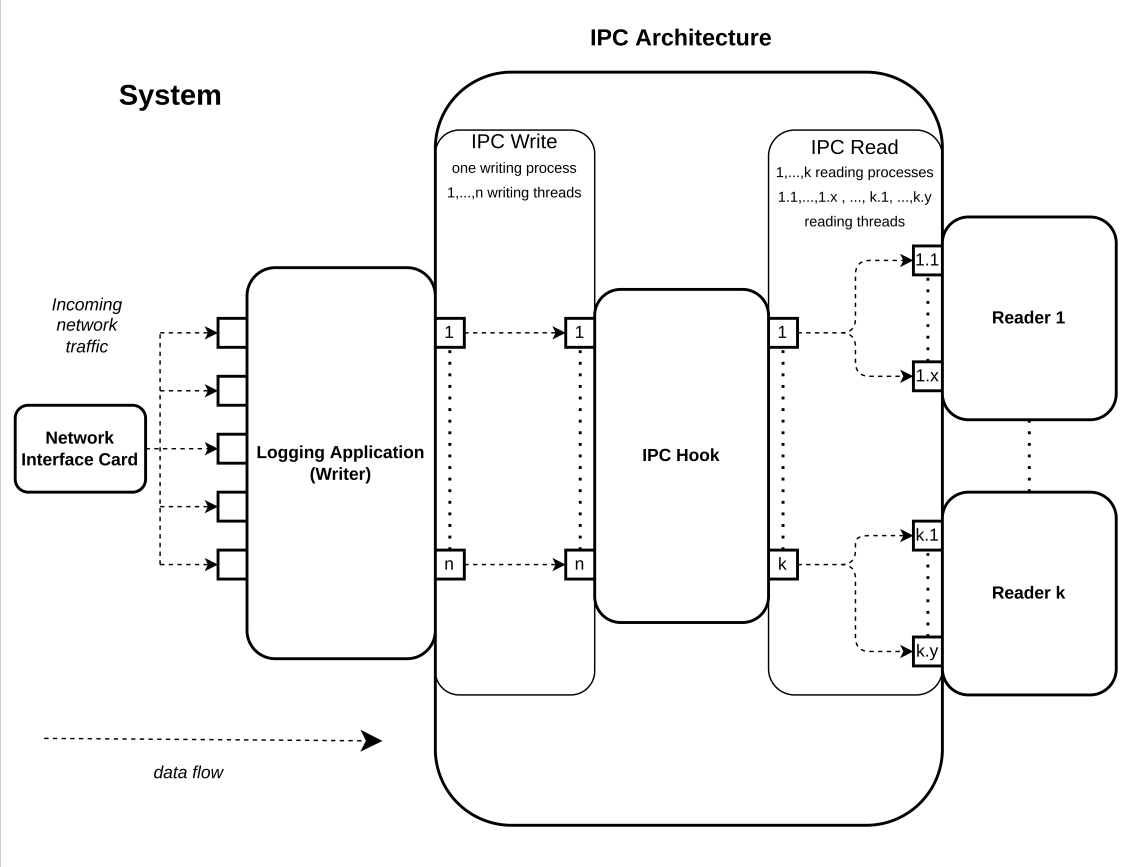


Figure 3.1: Abstract IPC architecture

IPC architecture is required to support a thread-safe write operation. Hence, it must be possible to write in parallel from *n* threads, without race conditions that could cause data corruption, or the need for additional thread synchronization on the writer side. On the reader side, the architecture must be able to support multiple reading processes, that each receive the entirety of the written messages and are able to read in parallel. To allow load balancing on the reader side as well, the read operation should also be capable of multi-threading. For the multi-threaded read within a reading process, the operation should enable a balanced an starvation-free distribution of messages to the calling threads.

Given the requirements and abstract architecture, the goal is to find a concrete implementation, that satisfied the proposed design.

3.3 Choice of IPC Type

During the development of this thesis, several **ICP!** mechanism where considered as suitable candidates for the proposed architecture. The initial objectives was to create multiple implementation that each use a different IPC type, in order to compare their performance empirically. To prevent the required time effort and scope of this thesis to get out of hand, the idea was abandoned in

favor of a single implementation. This has the additional benefit, of devoting more time to refine the concrete design and implement additional features.

All of the IPC types introduced in section 2.2.1

3.4 Ringbuffer API

The following section presents the design and implementation of the shared memory based architecture for the transmission of log messages.

3.4.1 Design

For the base structure of the shared memory design, a ringbuffer was chosen. A ringbuffer is a queue-like data structure, used to store multiple entries in a sequence. The entries are processed First in First out (FIFO) manner, hence data is being read in the same order that is was written. Ringbuffers are a commonly used structure to buffer data streams, especially network application []. They are usually realized as a fixed size array with additional pointers, that indicate the current location of readers and writers within the buffer. When the end of the buffer is reached during a read or write operation, the pointer wraps around to the begin of the buffer, threating the array, as if start and end where connected in a circular shape.

Figure **??** display the proposed ringbuffer architecture, adapted from the abstract design in Figure **??**. The shared memory region consists of a global header as well as a variable amount of segments, that each have their own segments header. Each segment constitutes a ringbuffer that is independent of the the other segments. The global header contains contains information that is global to the entire buffer and is used for initialization, when processes attach to the buffer. The buffer parameters within the global header are initializes by the processes creating the buffer and are supposed to be constant for the lifetime of the shared memory segment. The segment count variable indicates the amount of segments within the buffer. As illustrated in **??**, the number of segments is supposed to map 1:1 to the number of writing threads. Since each writing threads has their own ringbuffer, no synchronization for access to the write pointers is required. The individual entries within the segment ringbuffers are fixed size lines, the length of which in bytes is determined by the line size field in the global header. Each line corresponds to one log message, the same way a line in a logfile would. Limiting buffer entries to a fixed maximum size has some obvious disadvantages. The primary concern is inefficient use of memory, as the line size will have to be chosen to accommodate the largest possible log message, since multi line messages are not intended in the design. If log messages vary greatly in length, this may cause a lot of unused memory for shorter messages. The fixed size was chosen for a more convenient implementation of the overwrite feature, which is enable with the overwrite filed in the global header and will be covered in more detail later. Allowing for variable length entries is still an important consideration for further development of the design. The line count variable in the global header determines the number of entries each segment ringbuffer has. Making all segments equal in size implicitly assumes, that the distribution of log messages written is equal among the writing threads. However, variable size segments could be trivially added to the design, should a use case for that emerge. Finally, the global header contains a field for the number of allowed readers, as well as boolean

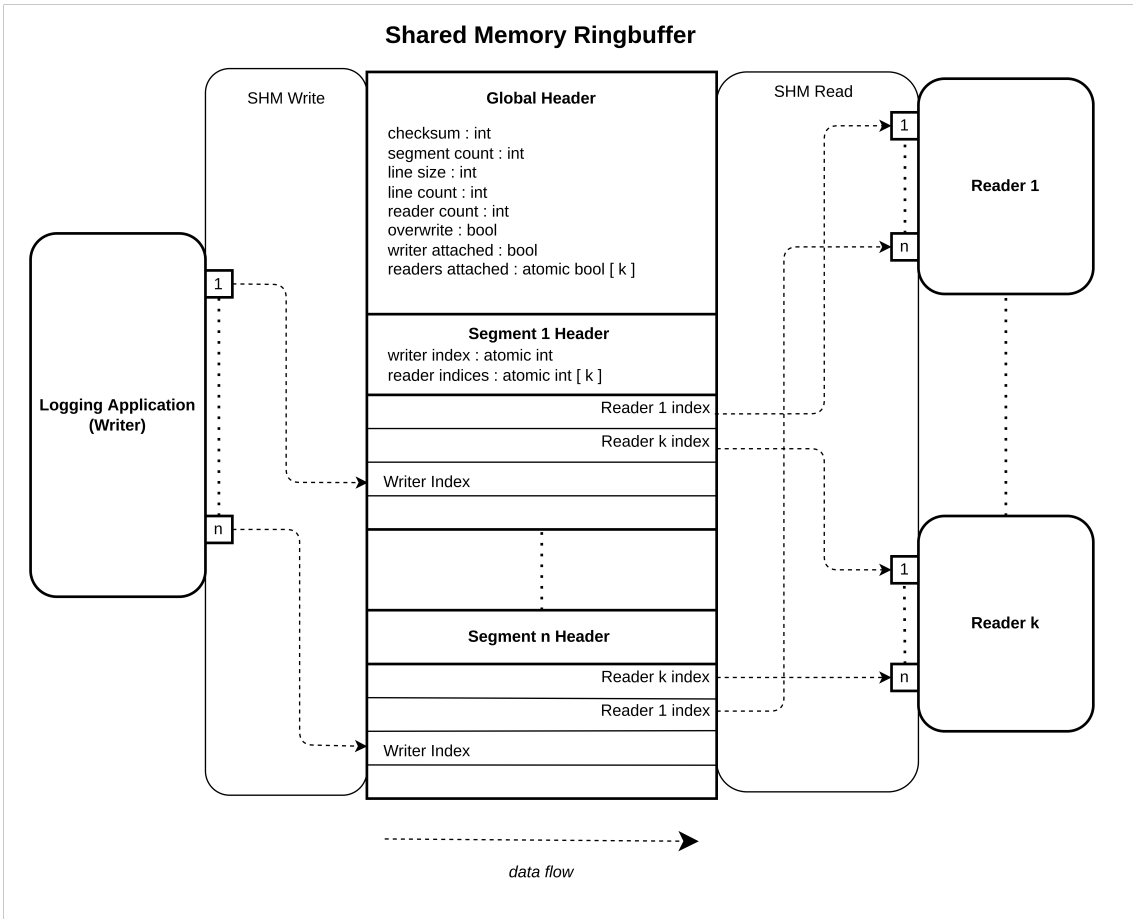


Figure 3.2: Architecture for the single-writer, multi-reader shared memory ringbuffer for the transmission of log messages.

fields, that indicate the attachment of the writer and the readers. Keeping the number of allowed readers fixed limits the ability of dynamically adding new readers, but makes the implementation more convenient. The attachment field are used for reference counting, of how many processes are attached to the buffer. When a reader attaches to the buffer, it will iterate through the readers attached array in the global header, in search of the first entry that that is not set to true. If an entry is found, the reader will set the value of the entry to true and the index of the entry becomes the readers reader id. Since writing to the attachment fields is implemented via atomic compare exchange operations, no race conditions exist for simultaneously attaching readers. Similarly, the writer uses the write attached field when attaching to the buffer.

Each segment of the buffer begins with its own header, that consist of a write index and an array of read indices, that is reader count elements long. The index corresponds to a line in the segment and is used to keep track of the current position of the reader or writer in the buffer, as well as to synchronize reader and writer. Both reader and write indices are implemented as atomic variables. Hence, read and write operation are executed like a single CPU instruction. This eliminates the possibility, that a processes can read the index of another processes while it

is being written and potentially obtain a corrupted value. Using atomic indices allows for lock free synchronization of reader and writer, which avoids the performance impact of having to switch threads between waiting and execution states that result from lock-based synchronization. However, atomic operations are also more costly compared to regular reads writes [1]. The lines of the segment begin below the segment header. The segment buffer capacity in bytes is given by line size "*" line count and the capacity of the entire buffer is given by segment capacity "*" segment count.

3.4.2 Implementation

In the following, the implementation of the design in figure **??** and relevant API functions is presented. The entire API was implemented as a static library in C. The associated header and source files are `shm_rbuf.h` and `shm_rbuf.c`, which can be found under src/lib in the Git repository for this thesis [16]. All API functions return numeric error codes, the meaning of which is also documented in `shm_rbuf.h`.

```
1 // Creates the ringbuffer or attaches to an existing one
2 int shmrbuf_init(union shmrbuf_arg_t * args,
3                   enum shmrbuf_role_t role);
4
5 // Detaches from the ringbuffer and removes the  memory
     segment, if no other process is attached
6 int shmrbuf_finalize(union shmrbuf_arg_t *,
7                       enum shmrbuf_role_t role);
```

Algorithm 3.1: Initialization and cleanup function for the shared memory ringbuffer.

Algorithm **??** displays the function signature for the initialization and finalizations functions. `shmrbuf_init` is used to create or attach to an existing shared memory buffer and has to be called before read or write operations on the buffer can be executed. `shmrbuf_finalize` is the corresponding function to detach or destroy the shared memory buffer and should be called upon ending the interaction with the buffer, for instance when exiting an application. `shmrbuf_finalize` will only destroy the shared memory segment, if no other process is attached. Otherwise the memory will simple be unmapped and the attachment field in the global header (as described in 3.4.1) is released. `shmrbuf_init` and `shmrbuf_finalize` can be called by both reader and writer processes, which is why a role has to be specified as an additional parameter in the function call. The role determines, what type of struct the union pointer shmrbuf_arg_t is being cast to, which contains further parameters used by the functions.

Algorithm **??** displays the struct, that is used by a writing processes, to store and pass related parameters to the API functions. In the current implementation, creation of a buffer can only be performed by a writing process. The reason for this is, that parameters such as the line size of segment count need to be tailored to the writing process. The writer creates a new buffer by parameterising a `shmrbuf_writer_arg_t` structure with the desired parameters for the buffer

[1] The performance evaluation of atomic vs lock based synchronization is not part of this thesis, but could be a consideration for a potential further development of this architecture.

```
1  struct shmrbuf_writer_arg_t
2  {
3      const char * shm_key;
4      uint16_t line_size;
5      uint32_t line_count;
6      uint8_t segment_count, reader_count;
7      struct shmrbuf_global_hdr_t * global_hdr;
8      struct shmrbuf_seg_whdr_t * segment_hdrs;
9      int flags, shm_id;
10 };
```

Algorithm 3.2: Structure to store writer parameters for the shared memory ringbuffer.

and calling `shmrbuf_init`. In addition to the already discussed parameters line size and line, segment and reader count, a key parameter is required. The key has to be a valid filepath within the file system and is used by System Vs shared memory API, to identify the shared segment[2]. If no shared memory segment referenced by the given path exists, a new segment of appropriate size will be allocated and mapped into the calling processes address space. Subsequently, its global header will be initialized with the parameters provided in the `shmrbuf_writer_arg_t` structure.

```
1  struct shmrbuf_global_hdr_t
2  {
3      uint32_t checksum;
4      uint8_t segment_count, reader_count;
5      uint16_t line_size;
6      uint32_t line_count;
7      bool overwrite;
8      atomic_bool writer_att , first_reader_att;
9  };
```

Algorithm 3.3: Structure to represent the global header of the shared memory ringbuffer.

Algorithm **??** displays the structure for the global header of the shared memory buffer. When the header has been initialized the checksum field is set to the sum over all constant header fields of the header. This is done as an additional security check for other processes attaching to the buffer. If the checksum is not correct, `shmrbuf_init` will fail, to avoid writing to or reading from potentially corrupted memory. After initializing the header, `shmrbuf_init` sets the indices of all segment headers to zero. As an alternative to creating a new buffer, a writer can also reattach to an existing buffer. When reattaching, the buffer parameters provided in `shmrbuf_writer_arg_t` are ignored. To enable reattachment, the `REATT` flags has to be specified in the flags field of

[2]The System V shared memory API [7] was used instead of the more modern POSIX API, since its support for memory allocation with huge pages was more convenient. Allocating the buffer with huge pages, should make read and write operations on a large buffer more efficient, since fewer entries are required in the translation lookaside buffer (TLB), making TLB misses less likely.

the shmrbuf_writer_arg_t, otherwise the call to `shmrbuf_init` will fail for an existing buffer. Reattachment is included in the API, so the writing application can be restarted (for instance for maintenance) without the need to destroy the shared memory segment, if readers are attached. The overwrite feature is also specified via the flag field by specifying the `OVWR` flag. Finally, `shmrbuf_init` initializes an array of segment header structures in `shmrbuf_writer_arg_t` and sets the `global_hdr` pointer to point to the start of the shared memory segment.

```
1 struct shmrbuf_seg_whdr_t
2 {
3     atomic_uint_fast32_t * write_index, * first_reader;
4     void * data;
5 };
```

Algorithm 3.4: Structure to store writer information for a segment of the shared memory ringbuffer.

Algorithm **??** displays the structure used in the segment header array. The structure does not represent the actual segment header the way its layed out in memory, but holds convenience pointers for a segment, that are used for writing operations.

For readers, attaching to an existing buffer is also done with a call to `shmrbuf_init`. This functions analogous the writer, with the exception that the shared memory segment pointed to by the provided path has to exist. Algorithm **??** displays the structure used for storing reader

```
1 // Reader parameters
2 struct shmrbuf_reader_arg_t
3 {
4     const char * shm_key;
5     int shm_id, flags;
6     uint8_t reader_id;
7     struct shmrbuf_global_hdr_t * global_hdr;
8     struct shmrbuf_seg_rhdr_t * segment_hdrs;
9 };
```

Algorithm 3.5: Structure to store reader parameters for the shared memory ringbuffer.

parameters. If the attachment to the shared memory segment and the checksum were successful, the reader will obtain a unique reader id, as described in section 3.4.1. If all reader slots are already occupied, the `shmrbuf_init` call fails. Subsequently, `shmrbuf_init` will initialize the pointers to the global header and the array of segment headers structures in the provided `shmrbuf_reader_arg_t` structure. Readers have their own structure for storing segment header information, which is displayed in Algorithm **??**.

The structure includes a pointer to the index of the reader within the segment header. Which read index of a segment header is assigned to a reader, is determined by their reader id. Additionally, the structure contains a mutex lock, that is used to synchronize access to the segment when reding with multiple threads. After a successful call to `shmrbuf_init`, both reader an writer can start using their respective APIs for the buffer.

```
1  struct shmrbuf_seg_rhdr_t
2  {
3      atomic_uint_fast32_t * write_index, * read_index;
4      pthread_mutex_t segment_lock;
5      void * data;
6  };
```

Algorithm 3.6: Structure to store reader information for a segment of the shared memory ringbuffer.

3.4.3 Write API

```
1  // Writes a single line to a segment
2  int shmrbuf_write(struct shmrbuf_writer_arg_t * args,
3                    void * src, uint16_t wsize,
4                    uint8_t segment_id);
5
6  // Writes multiple lines to a segment
7  int shmrbuf_writev(struct shmrbuf_writer_arg_t * args,
8                     struct iovec * iovecs,
9                     uint16_t vsize,
10                    uint8_t segment_id);
```

Algorithm 3.7: Write API for the shared memory ringbuffer.

Algorithm **??** displays the write API for the shared memory ringbuffer. `shmrbuf_write` writes a single log message to a specified segment within the buffer. The segment being written to has to be reference with an id, which is simply its index within $\{0..n\}$, where n is the number of segments. The src argument is a pointer to the string that should be written to the buffer and `wsize` specifies the length of the string. If `wsize` exceeds the line size specified in the global header of the buffer, the write operation will fail with a size error. When issuing a call to `shmrbuf_write` without overwrite, the the function will atomically load all reader indices in the segment header and check their distance to the write index. If no reader is directly ahead of the write index, the line will be written to the buffer and the writer index will be atomically updated to its new position, otherwise, the function call returns with an error. If the overwrite operation is specified in the header, synchronization between writer and reader is effectively disabled and the write operation will conclude, regardless of the readers positions. This incurs the risk, that a reader and writer simultaneously access the same line, which may potentially corrupt the data being read. An advantage of overwrite is, that the writer avoids costly atomic operations and also cant be blocked by a slow or stale reader. The performance of the overwrite feature will be evaluated in section 4. The write API provides a seconds write function in `shmrbuf_writev`. Instead of writing a single line, the function receives an an array of `iovec` structures. The `iovec` structure defined in `struct_iovec.h` within the standard C library, is a generic container for vectored io operations. It contains a void pointer to a data field and a variable to specify the length of the

data in bytes. shmrbuf_writev works analogous to `shmrbuf_write`, with the only difference that `vsize` lines referenced the corresponding `iovec` structure in iovecs will be written to the buffer. Performance inspection of `shmrbuf_write` with the benchmarking tool perf [17] revealed, that about 75% of the execution time is being spent for executing the atomic operations. Hence, using a vectored write operation should improve performance, as the synchronization only has to be applied once for multiple log messages. For the sake of performance, neither write function implements synchronization for the write index. Calling a write operation on the same segment from two different threads, is therefore not thread-safe.

3.4.4 Read API

Algorithm **??** presents the functions of the reader API.

```
1  // Reads a single line from a segment
2  int shmrbuf_read(struct shmrbuf_reader_arg_t * args,
3                   void * rbuf,
4                   uint16_t bufsize,
5                   uint8_t segment_id);
6
7  // Reads multiple lines from a segment
8  int shmrbuf_readv(struct shmrbuf_reader_arg_t * args,
9                    struct iovec * iovecs,
10                   uint16_t vsize,
11                   uint16_t bufsize,
12                   uint8_t segment_id);
13
14 // Reads a line from a segment ouf of a specified range.
15 int shmrbuf_read_rng(struct shmrbuf_reader_arg_t * args,
16                      void * rbuf,
17                      uint16_t bufsize,
18                      uint8_t lower,
19                      uint8_t upper,
20                      bool * wsteal);
21
22 // Reads multiple lines from a range of segments
23 int shmrbuf_readv_rng(struct shmrbuf_reader_arg_t * args,
24                       struct iovec * iovecs,
25                       uint16_t vsize,
26                       uint16_t bufsize,
27                       uint8_t lower,
28                       uint8_t upper,
29                       uint16_t * wsteal);
```

Algorithm 3.8: Read API for the shared memory ringbuffer.

shmrbuf_read is the corresponding read operation to shmrbuf_write. It reads a single from the specified segment into the provided buffer. If the provided buffer size is exceeded by the size of the line, the read call fails, hence, the buffer should always at least have maximum line size, specified in the global header. Upon a call to shmrbuf_read, the function atomically loads the write index for the segment. If the position of the write index differs from that of the readers read index, the line pointed to by the reader index will be copied to the external buffer and the read index will be atomically updated to its new position. On success, shmrbuf_read returns the (maximum) line size or zero, if the buffer is empty [3]. Analogous to the write API, a vectored read function exists in shmrbuf_readv, which should incur the same performance benefits. Instead of reading a single line, vsize lines are copied to the buffers specified by in corresponding iovec structures in iovecs. Unlike the write API, all read operations called on the same segment by threads within the same processes are thread safe. At the beginning of each read operation, the mutex lock in the corresponding shmrbuf_seg_rhdr_t structure is claimed and release after the read index has been updated. For performance reasons, frequent locking of a segment by different reading threads should be avoided and threads should ideally be assigned segments that they access exclusively. The read API contains to further functions. shmrbuf_read_rng is a convenience wrapper around shmrbuf_read. It its useful for the case, that the reading application uses fewer reading threads than there are segments in the buffer. shmrbuf_read_rng allow the specification a segment range through the upper and lower parameters. When calling shmrbuf_read_rng, the function will iterate over all segments in the range in a round-robin fashion and perform a read call, until the first non empty segment is found or one cycle is completed. The iteration index is persistent across function calls, ensuring that subsequent calls will continue the round-robin at the same position. This allows starvation-free reading from the specified segment range. The boolean pointer wsteal allows for the option of workload stealing. If wsteal is non null and all segments within the specified range are empty, the function will additionally iterate over all segments outside of the specified range until a line has been read successfully or all segments have been checked. If a line was read from outside the specified range, wsteal will be set to true. The intention of the workload stealing feature is to mitigate scenarios, where writing operation are not equally distributed across segments. If a reader, for instance, assigns two reading threads to one segments each, but only one of the segments is being written to, then one reading thread would be idle, while the other has to handle all reading operations. Workload stealing mitigates this, by having threads read across their assigned boundaries, if they would otherwise be idle. shmrbuf_readv_rng is the last function in the read API and corresponds to shmrbuf_read_rng with the only difference, that multiple lines can be read from each segment, up to the number specified by vsize. The wsteal parameter is also an integer instead of a boolean pointer and will be set to the number of lines read from outside the specified range, if wsteal is non null.

[3] In its current implementation, the buffer allows any type of binary data to be copied to a line. This is a problem for determining the actual length of a string in the line buffer, since the terminating zero byte could still be part of the payload. Therefore, the maximum line length is always returned. Restricting the buffer to non zero characters, or moving to a variable line length design, would enable the function to determine the length of the string contained in the line buffer, making the return value more meaningful.

3.5 Proof-of-Concept IPS

The following section presents the design and implementation of the proof of concept IPS, for testing the IPC architecture covered in the past sections. In order to be comparable, the IPS will be closely modelled after Fail2ban. The goal is however not to fully reimplement Fail2bans full set of feature, as that would go beyond the scope of this thesis. Instead, a minimal set of features will be supported, the allows for the replication of the experiment covered in section 2.1.1. The requirements for the **Poc!** (**Poc!**) are as follows:

- Ability to monitor an application log, via traditional logfile parsing and the new IPC solution.
- Ability to parse log messages with a customly definable regular expression.
- Ability to create filter rules for a clients IP addresses for both Internet Protocol Version 4 (IPv4) & IPv6.
- Support for a ban limit, i.e. a customly definable number of matches per client before a ban is executed.
- Support for a ban time, i.e. a customly definable duration for the ban of a clients IP address.

Figure 3.3 illustrates the proposed design for the Proof of Concept (PoC), which will from hereon be referred to as Simplefail2ban. The source file for the implementation of Simplefail2ban is `simplefailban.c` in the `src/programs` directory of the thesis Git repository [16].

The functionality of the application will be separated into two classes of threads. A variable number of "banning" threads and a single "unbanning" thread. The banning threads are tasked with monitoring the application log. The routinely call a read function, to check if new messages have been added to the log. The read function varies, depending on the type of IPC architecture being used. For multi-threaded use, the read function has to be thread-safe and should ideally be lock free, to allow for optimal performance. When using the shared memory ringbuffer as the source for log messages, `shmrbuf_readv_rng` is used, where the buffer segments are equally distributed among the banning threads. For traditional logfile parsing, a custom read function was implemented, on the basis of the asynchronous Input / Output (IO) library liburing introduced in 2.3.2.

?? displays the function signatures. `uring_getline` was modelled after the getline function from the `stdio.h` library within the C standard library [18]. The function is passed a pointer to a `file_io_t` structure which contains file descriptor for the designated file, as well two buffers for the data being read. When the function is called, both buffers are checked for lines from an earlier function call. If both buffers are empty, a liburing based read operation is initiated. The read data is then tokenized into lines via a linear search for newline characters. If the read operation fills the entire buffer, another read operation is scheduled for the second buffer. The return of the read operation is not awaited and will only be checked on a subsequent function call. The idea is, that, in anticipation of subsequent function calls, data is being asynchronously read ahead, so that for most function calls, the function can resort to the buffer instead of reading from the actual file. This should hopefully be more performant than regular blocking calls to `read`. The file offset for the read operation is tracked in the `file_io_t` structure and updated

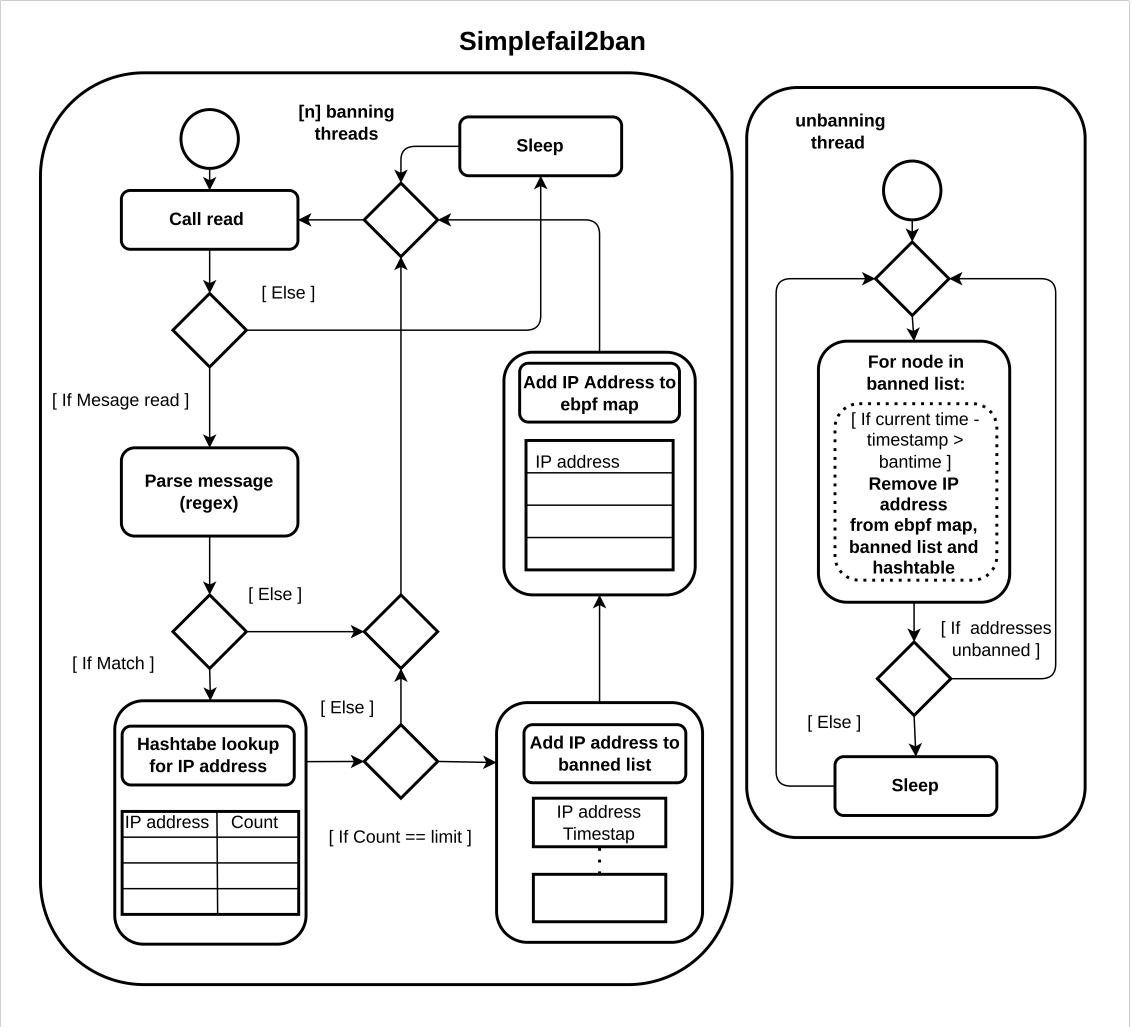


Figure 3.3: Activity diagram for the proof-of-concept IPS implementation. A variable number of "banning threads" receive log messages from a host and parse them with a predefined regular expressions. For messages that match the expression, the clients IP address is extracted from the log message and added to a hashtable, that keeps count of the number of matches per address. If the count reaches the configured limit, the address is added to the list of banned addresses with a current timestamp and inserted into the eBPF map. One "unbanning thread" routinely iterates through the banned list and checks, if a clients bantime hast elapsed. Clients with an elapsed bantime are removed from the eBPF map, banned list and hashtable.

```
1 // Reads a single line from a file (buffered)
2 int uring_getline(struct file_io_t * fio_arg,
3                   char ** lineptr);
4
5 // Reads multiple lines from a file (buffered)
6 int uring_getlines(struct file_io_t * fio_arg,
7                    struct iovec * ivoecs,
8                    uint16_t vsize,
9                    uint16_t bufsize);
```

Algorithm 3.9: Function signatures for the io_uring based getline functions.

after each read. `uring_getlines` works analogous to `uring_getline`, with the difference that up to `vsize` lines can be read in a single function call. The lines and their size in byte are copied to the corresponding `iovec` structure in `iovecs`. `uring_getline` and `uring_getlines` are not build for thread-safety, hence, the logfile based version of Simplefail2ban only supports single threaded monitoring[4].

If a banning thread has successfully read one or multiple log messages, the messages are then matched against a regular expressions, which can be defined at the start of the application. For REGEX-matching the regular expressions engine Hyperscan, introduced in section 2.3.1, is used. Hyperscan is used in multi-pattern matching mode, scanning for both the provided REGEX as well as expressions for IPv4 and IPv6, to extract the clients IP address from the log message. If the message was successfully matched against the provided REGEX, the clients IP address identified by the IP-REGEX will be translated to its binary form for further processing. If the scan does not find a match, the processing moves on to the next log message. Alternatively, Simplefail2ban can also be configured to directly translate the entire log message to an IP address, without regular expression matching. This is not explicitly referenced in 3.3 and was only implemented as reference, to measure the impact of REGEX-matching on the overall performance of the application.

Once a clients binary IP address has been determined (either through successful matching or direct translation), a hash table lookup for the address is performed. The purpose if this is to determine, if a client has reached the ban limit, which is configured at the start of the application. The hash table stores key value pairs, where the key can be a IPv4 or IPv6 address in binary form and the value is a counter, indicating the number of times the key has been queried. A custom hash table was implemented as a static library, the sources files for which are `ip_hashtable.h` and `ip_hashtable.c` in the `src/lib` directory of the thesis Git repository [16].

?? displays the structure used for storing a single entry in the hash table. The base table consist of an array of bins, with a default size of 6000011[5]. When an ip address is inserted into the table, an index within the array is determined by calculated the value of a hash function modulo the size of the array. The address and its associated counter value are then stored in the bin at the determined index, if it is empty. The hash function used for the implementation is spookyc,

[4] It its unclear, wether multi-threaded file reading would provide a performance benefit, since the read operations would have to be serialized by the operating system. Hence the decision to not support multi-threading.

[5] The array size was purposefully chosen as a prime number, to allow for a more equal spread among the bins.

```
1  // Struct to store a single hashtable entry
2  struct ip_hashbin_t
3  {
4     void * key;
5     int domain;
6     uint32_t count;
7     pthread_mutex_t lock;
8     struct ip_hashbin_t * next;
9
10 };
```

Algorithm 3.10: Structure for storing a single entry in the IP hash table. The `key` pointer points to the binary address, the size of which depends on wether the domain value is `AF_INET` or `AF_INET6`.

which is a C implementation of the spooky hash function by Bob Jenkins [19]. If the bin at the determined is not empty, a collision occurs. Collisions are handled via a linked list, where the bin in the base array constitutes the first node of the list.

Number of Insertions	Collisions IPv4 [%]	Collisions IPv6 [%]	Collisions IPv4 & IPv6 [%]
65534	5.33	5.29	5.28
131068	10.17	10.17	10.15
600011	36.81	36.77	36.8

Table 3.1: Percentage of hash collisions by key type for different numbers of insertions. The results were obtained with `hashfunc_benchmark.c` in the `src/utilities` directory of the thesis Git repository [16].

Table **??** presents experimental results for the collisions performance of the hash function. There appear to be no significant differences in collision behaviors between IPv4 and IPv6 addresses. For 6000011 insertions (size of the base table), a little more than a third of all insertions resulted in a collision. The hash table is global to the application, since different banning threads may handle log messages for the same client and need to synchronize their match count. To ensure thread-safe operations on the hash table, locking via mutexes is used. The granularity of the locking is at bin level, hence, parallel operations on the table are possible if different bins are accessed.

If the hash table lookup returns a count that is equal to the ban limit, the banning thread facilitates a ban action, analogous to Fail2ban. To implement IP address filtering, the eBPF program developed by Florian Mikolajczak [5] as described in 2.1.1, was used. The program can be loaded onto an network interface and is executed event based on incoming packets[6]. To determine which packets should be dropped, the program used maps containing IP addresses, which are pinned to the eBPF file system. At the start of Simplefail2ban, the eBPF program will be loaded onto a

[6]For a more through and detailed explanation, see [5]

configured interface. The function used for loading and unloading the eBPF program and corresponding maps are slightly adapted from the implementation by Florian Mikolajczak and can be found in `ebpf_helpers.h` and `ebpf_helpers.c` in the `src/lib` directory of the thesis Git repository [16]. The ban action executed by the banning thread consists of adding a clients binary IP address to the corresponding eBPF map. There are two distinct maps being used for IPv4 and IPv6 addresses. Once the address has been added to the map, the eBPF program will drop all incoming packets from that address, for as long as the address is contained in the map. Additionally, the banning thread adds the IP address to a linked list (referred to as banned list in 3.3) together with a current timestamp. The linked list is used to store client that are currently banned. The source files for the linked list implementation are `ip_llist.h` and `ip_hllist.c` and can be found in the `src/lib` directory of the thesis Git repository [16]. The unbanning thread will routinely iterate over the list and checked the amount of time that has passed since the timestamp. If the configured ban time has elapsed, the unbanning thread will unban the client. To unban a client, the unbanning thread removes its entry in the linked list as well as the entry in the eBPF map. Finally, the address entry in the hash table is removed. This resets the matching count, so a client can be banned, if the ban limit is reached again.

3.6 Test Application

To evaluate the IPC architecture in conjunction with Simplefail2ban, a test application is needed, which utilizes the IPC API to transmit log messages. For this purpose, a test server was developed, the source file for which is `udp_server.c` in the `src/programs` directory of the thesis Git repository [16]. Alternatively, the IPC architecture could have been integrated into a real application, like the BIND server used in the previous Fail2ban measurements by Florian Mikolajczak. While this would have provided a more realistic basis for the evaluation, the associated implementation effort was deemed beyond the scope of this thesis. The test server serves as a stand in for a real UDP based application such as BIND. It listens on a configurable port, and replies to incoming packets with a single one byte payload UDP packet. Based on the first payload byte of the received request, the server decides wether to write a log string for the requesting client[7]. The log string contains the current date, time and IP address of the client, as well as a descriptive message and is written to either a logfile or the shared memory ringbuffer. Alternatively, the test server can be configured to only log a clients IP address. The test server uses multiple threads to listen for incoming packets and is designed to handle a large amount of requests and logging operation per second, in order to not be a bottleneck for the evaluation of Simplefail2ban. Perf [17] evaluation of the test server revealed, that a about 20% of the execution time on the application side was spent translating IP addresses to string form. To facilitates fast creation of log strings, a custom function was written for the transformation of a binary IP address to string form.

Table **??** summaries the evaluation results for the custom function `ip_to_str` and the standard library function `inet_ntop`. `ip_to_str` is on average about 6 times faster for translating IPv4 addresses and about 7 times for IPv6. Since the test server can write up to a million log messages per second, overall performance should be improved by the use of `ip_to_str`[8].

[7] The byte to trigger logging can be determined with the `INVALID_PAYLOAD` macro in `udp_server.c`.

[8] This could be further evaluated by the comparing overall difference of the server for both functions, but I unfortunately did not have the time to implement further test.

Function	Execution Time IPv4 [Seconds]	Execution Time IPv6 [Seconds]
`inet_ntop`	1.29	3.98
`ip_to_str`	0.21	0.53

Table 3.2: Performance evaluation for binary IP address to string conversion. The evaluated functions are `inet_ntop` from `arpa/inet.h` in the C standard library and a custom function `ip_to_str`. The corresponding source files are `ip_to_str.h` and `ip_to_str.c` in the `src/lib` directory and the evaluation was conducted with `ip_string_benchmark.c` in the `src/utilities` directory of the thesis Git repository [16]

3.7 Other Applications

Two other applications where developed as part of this thesis. To test the multi reader capability of the proposed shared memory architecture, an application serving as an additional reader was implemented. The application models a log aggregator such as Logstash (discussed in 2.2.1) and writes the log messages contained in the buffer to a configurable logfile. The source file for this application is `simplelogstash.c` in the `src/utilities` directory of the thesis Git repository [16]. The second application is a utility for inspection and debugging the shared memory ringbuffer. It allows the inspection of the header as well as the display of load statistics on the individual segments and the entire buffer. The source file is `poll_rbuf.c` in the `src/utilities` directory of the thesis Git repository [16].

4 Evaluation

The following section presents the performance evaluation of the proof of concept IPS Simplefail2ban, in conjunction with the test application. First, an overview over the test environment and the experimental design is given and the Fail2Ban performance issues discovered by Florian Mikolajczak are replicated for the test application. Subsequently, the result for Simplefail2ban in both the logfile and shared memory based variants are presented. The section concludes with an evaluation of different features of the shared memory implementation.

4.1 Test Environment

To conduct the evaluation, two machines with an identical hardware and software configuration where used. The specific hardware and software version are listed in Table 4.1. One machine served as the dut, running the test application, as well as Fail2Ban / Simplefail2ban. The second machine was used for traffic generation with trex.

Table 4.1: Hardware and Software parameters for the test environment. The measurements where conducted on two identical machines, where one machines served as the Device under Test (DUT), running the test application and Fail2Ban / Simplefail2ban, while the other generated test traffic with trex.

Hardware	
CPU	Intel(R) Xeon(R) Silver 4314 CPU @ 2.40GHz
NIC	Mellanox ConnectX-6 100GbE
RAM	128GB
Software	
OS	Debian 11
Kernel	6.1.0-0
NIC Driver	mlx5_core, 6.1.0-0
Fail2Ban	0.11.2
TRex	3.02

4.2 Experimental Design

The experimental design is closely oriented towards experiment 1 conducted by Florian Mikolajczak in [5], in order to provide comparable results. In the experiment, a **DOS!** attack on a server under a normal workload is simulated, through a small stream of valid a request and a significantly larger stream of invalid request. The original experiment used a BIND9 DNS server as the target of the **DOS!** attack, which triggered log messages for clients exceeding a certain rate limit. For the following evaluation, the test application introduced in **??** will serve at the target, differentiating valid and invalid request by the first byte of the sent payload[1]. While this way of determining the validity of a request is not necessarily realistic, it is efficient and allows the test application to quickly produce log messages. The test subject of the experiment For invalid request, a log messages containing the current date, time and client IP address is written to either a logfile or the shared memory ringbuffer.

The goal of this evaluation is to test, how the proof of concept IPS implementation performs in the test scenario described above.

Following the criteria

4.3 Fail2ban Replication Measurements

4.4 Simplefail2ban, Logfile Measurements

4.5 Simplefail2ban, Shared Memory Measurements

4.6 Shared Memory Feature Measurements

[1] Request are considered invalid, if the first byte if the payload has the unsigned value 42 (ASCII Letter B)

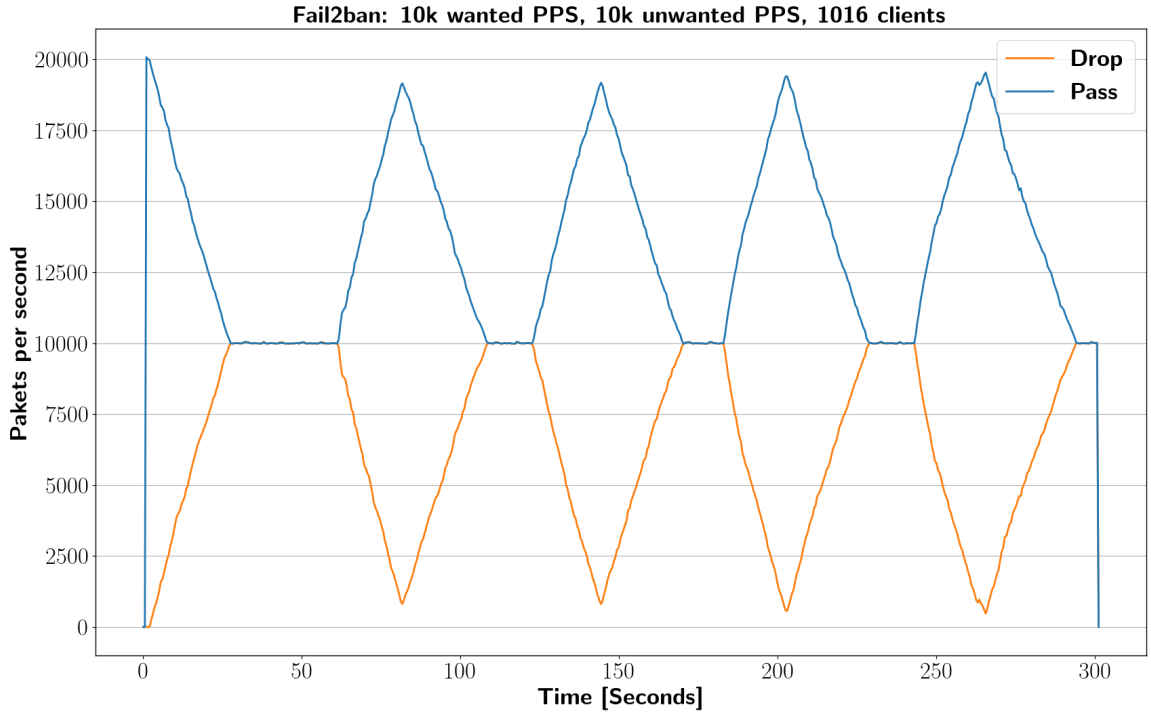


Figure 4.1: Abstract architecture

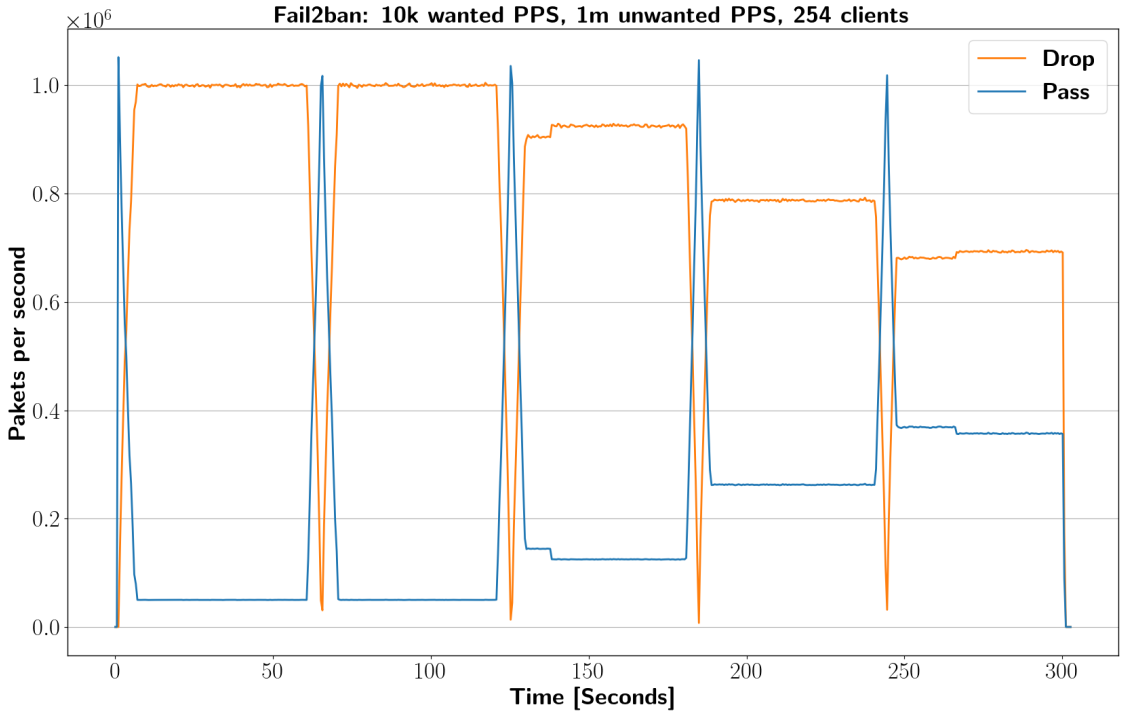
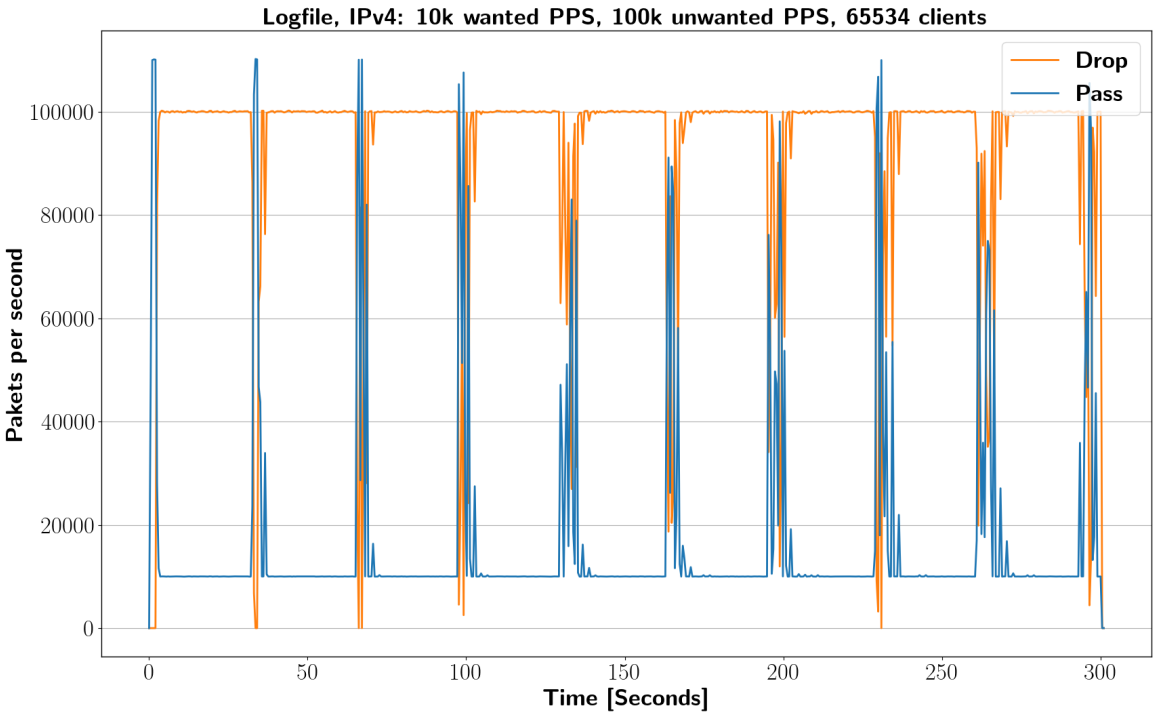
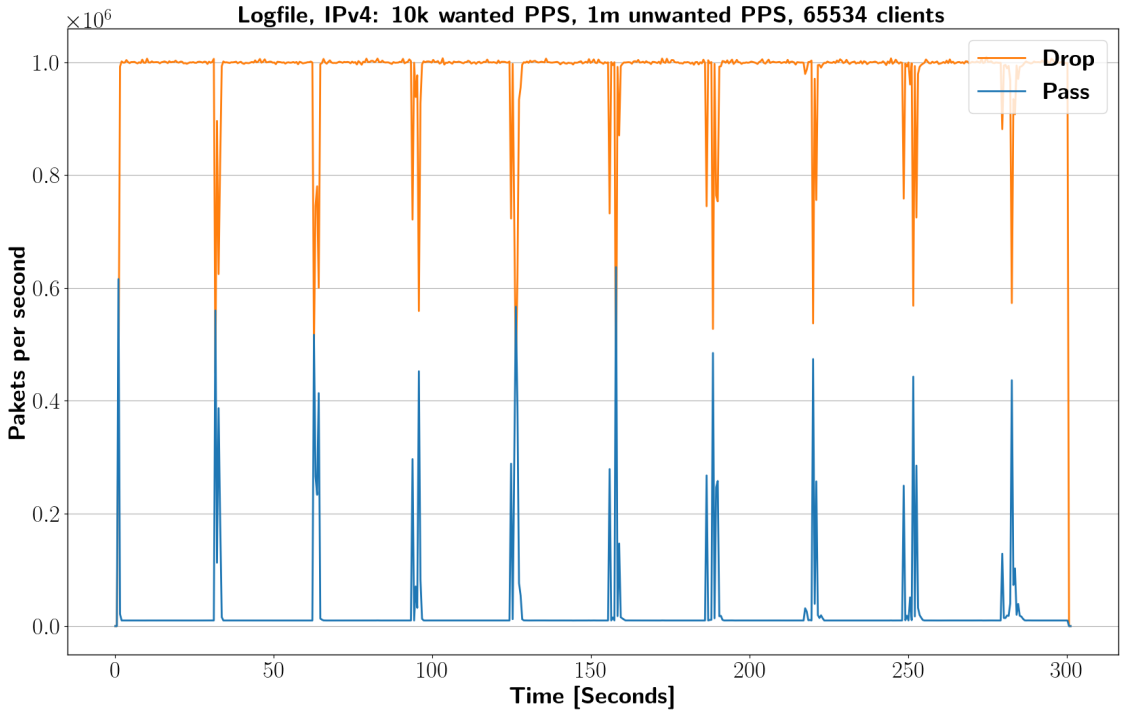


Figure 4.2: Abstract architecture



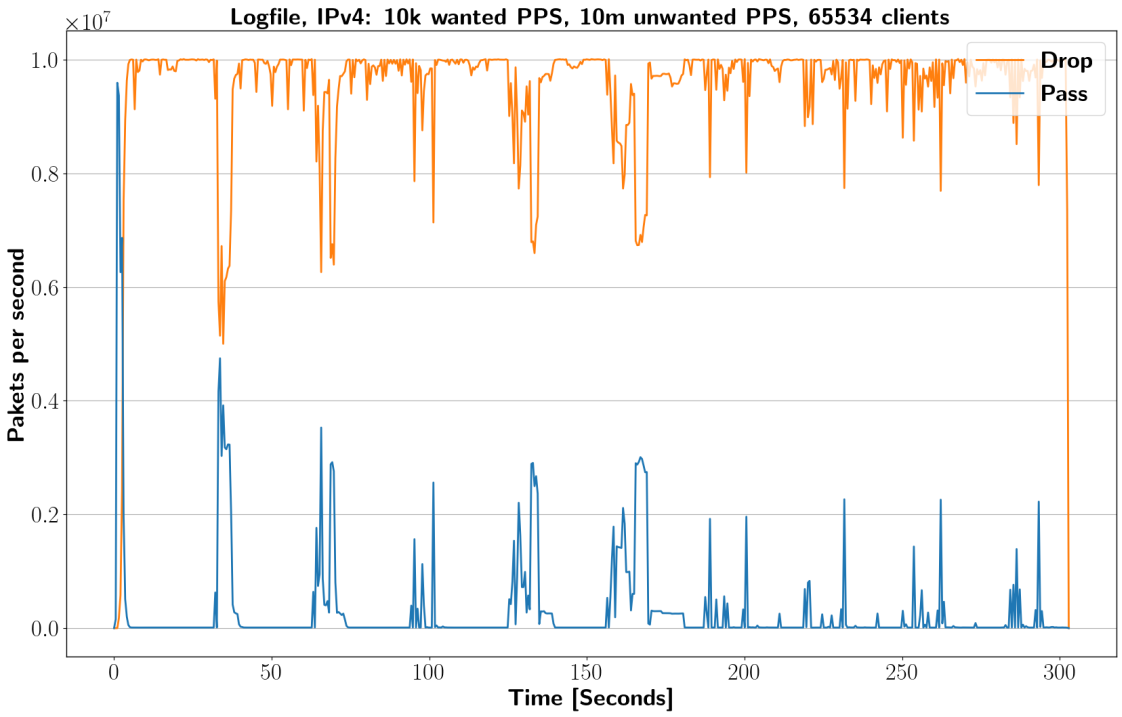
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
33	27.93	99.59	2.07	10.51

Figure 4.3: Some text



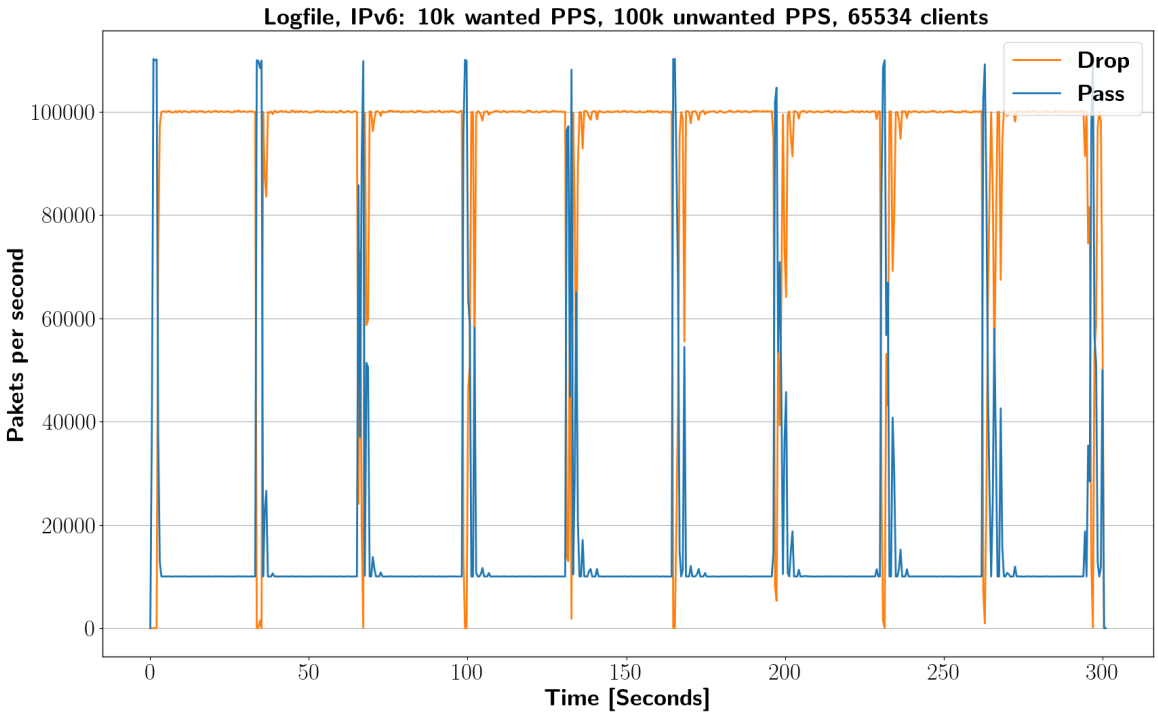
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
303	294.43	98.79	4.7	14.99

Figure 4.4: Some text



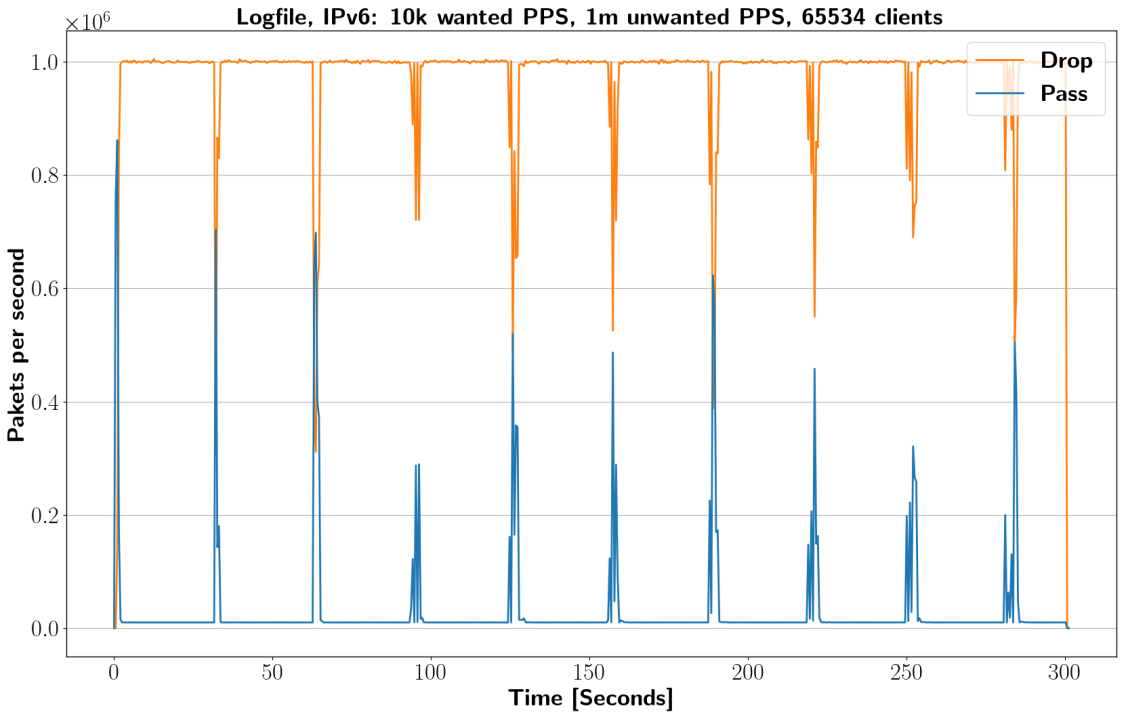
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
2986.89	2884.14	96.72	34.39	83.72

Figure 4.5: Some text



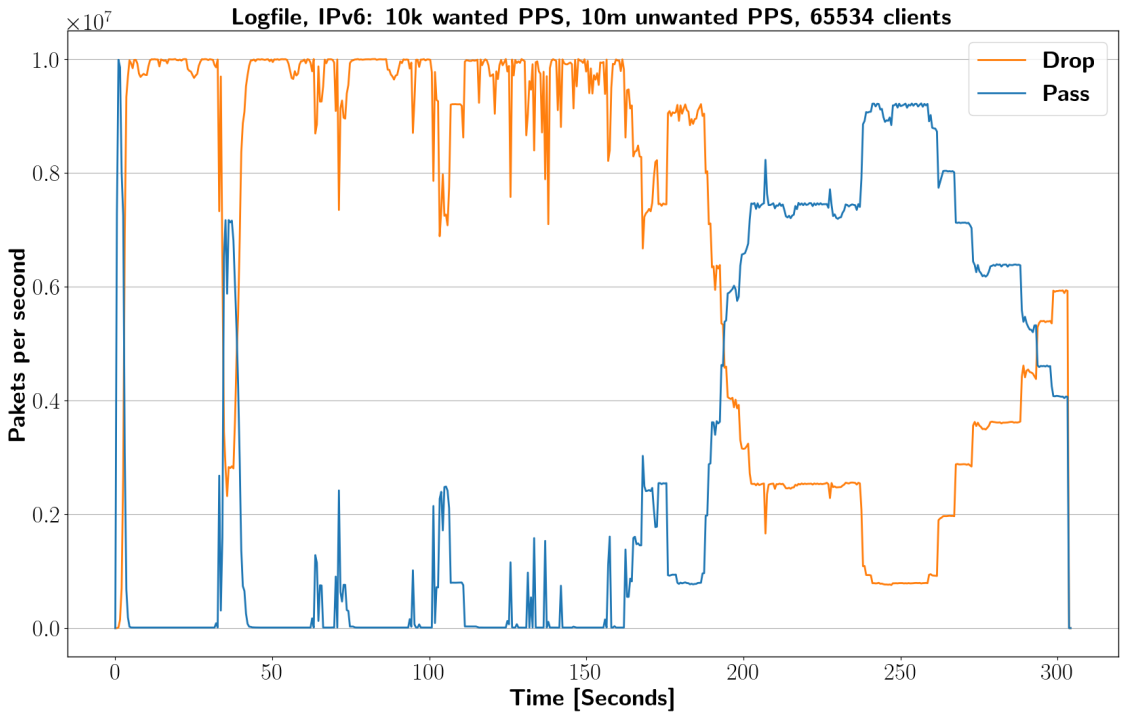
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
33	27.96	99.67	2.04	11.75

Figure 4.6: Some text



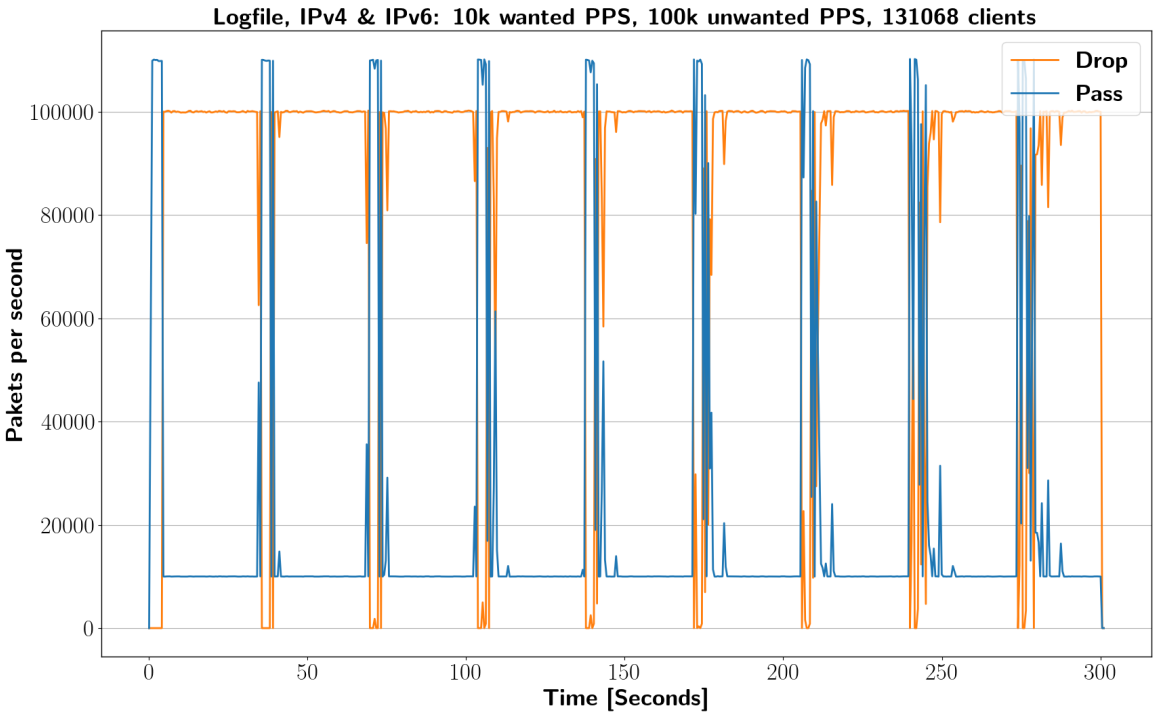
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
303	293.49	98.48	5.58	21.08

Figure 4.7: Some text



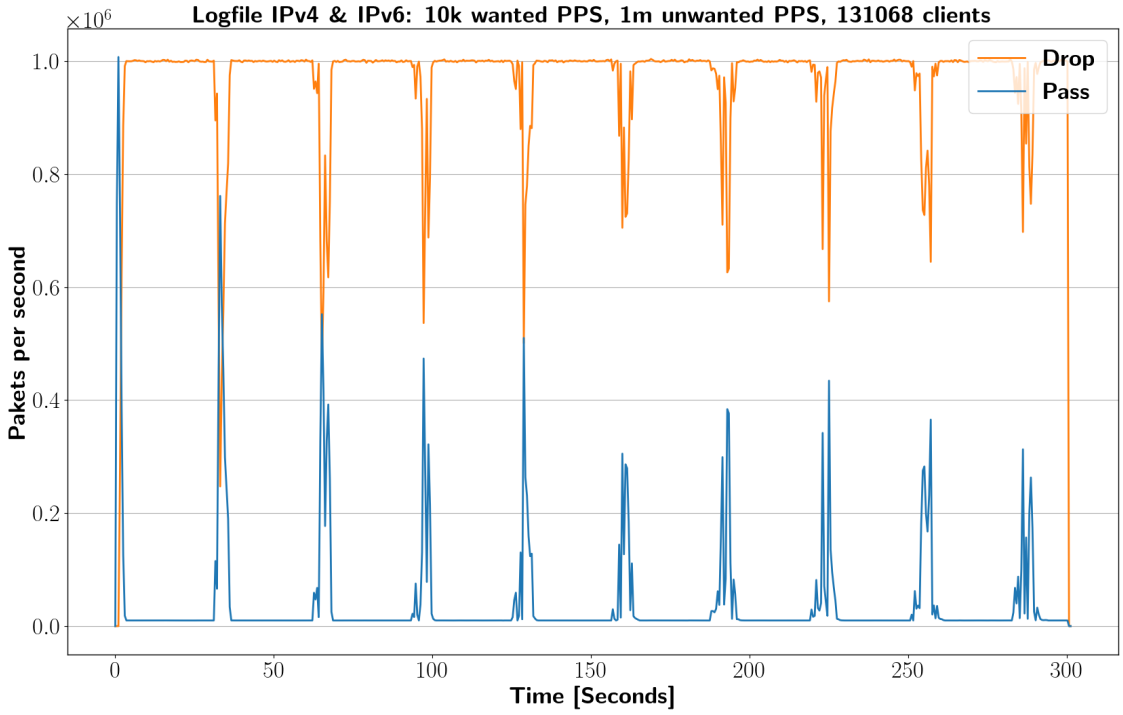
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
2992.43	2065.28	69.13	205.4	346.73

Figure 4.8: Some text



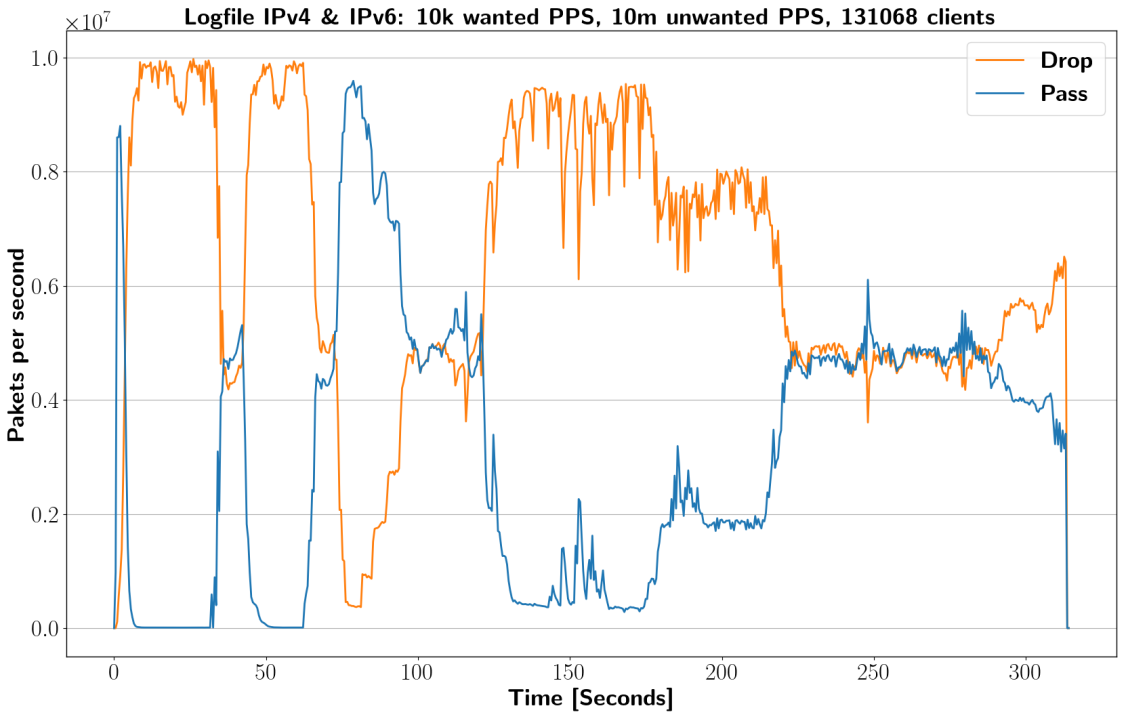
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
33	26.45	99.96	3.55	18.47

Figure 4.9: Some text



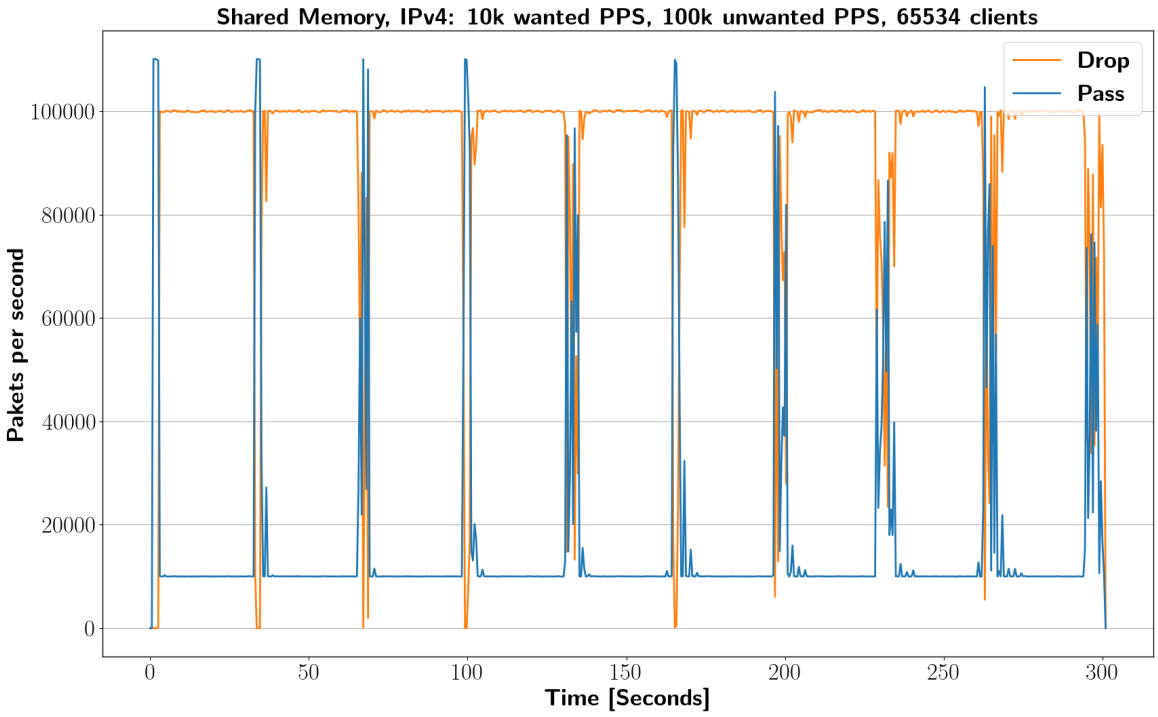
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
303	290.47	98.11	8.5	32.94

Figure 4.10: Some text



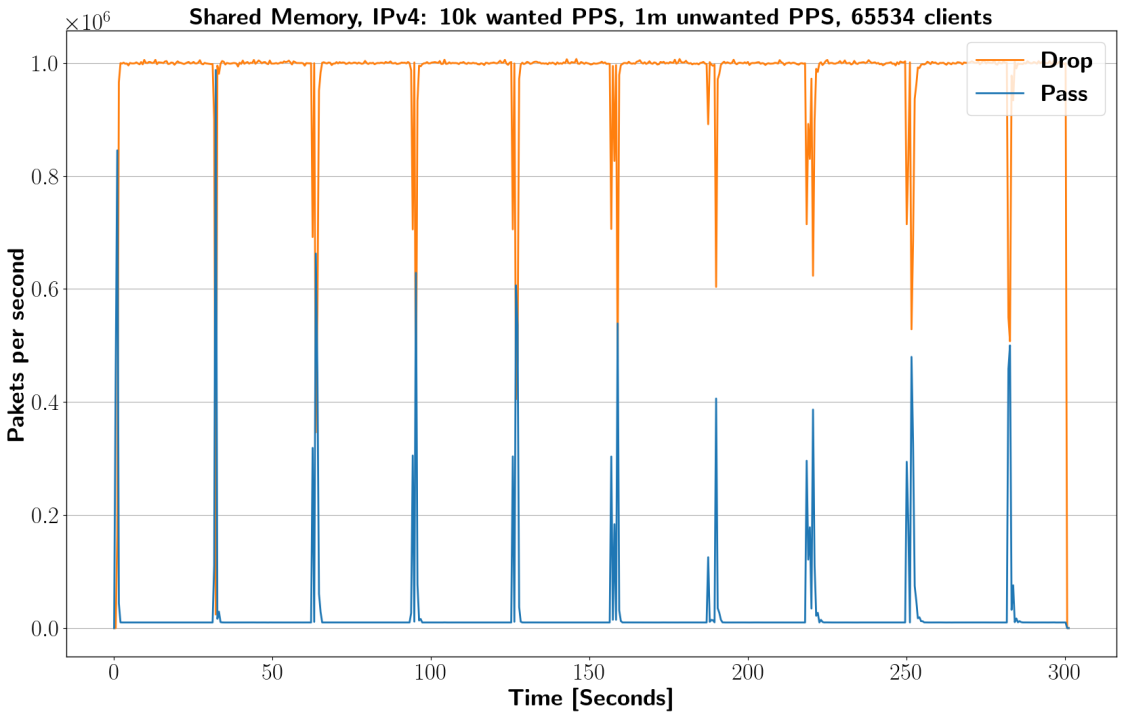
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
2993.57	2014.77	67.46	213.63	382.19

Figure 4.11: Some text



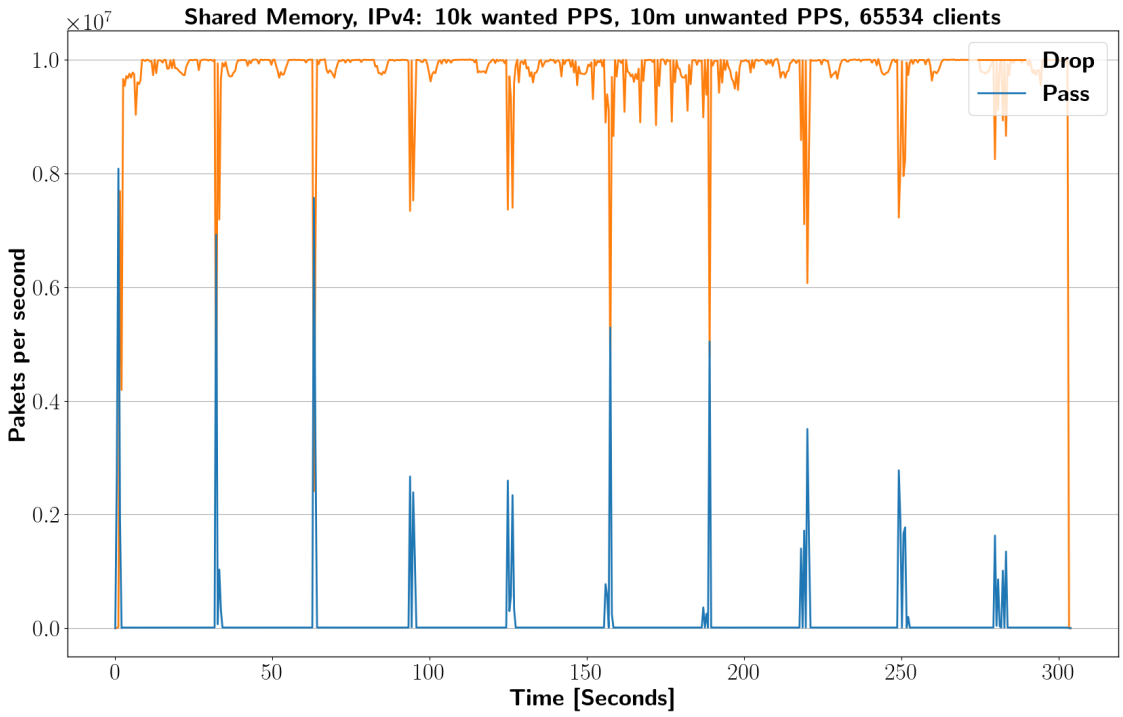
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
33	27.95	99.64	2.05	13.68

Figure 4.12: Some text



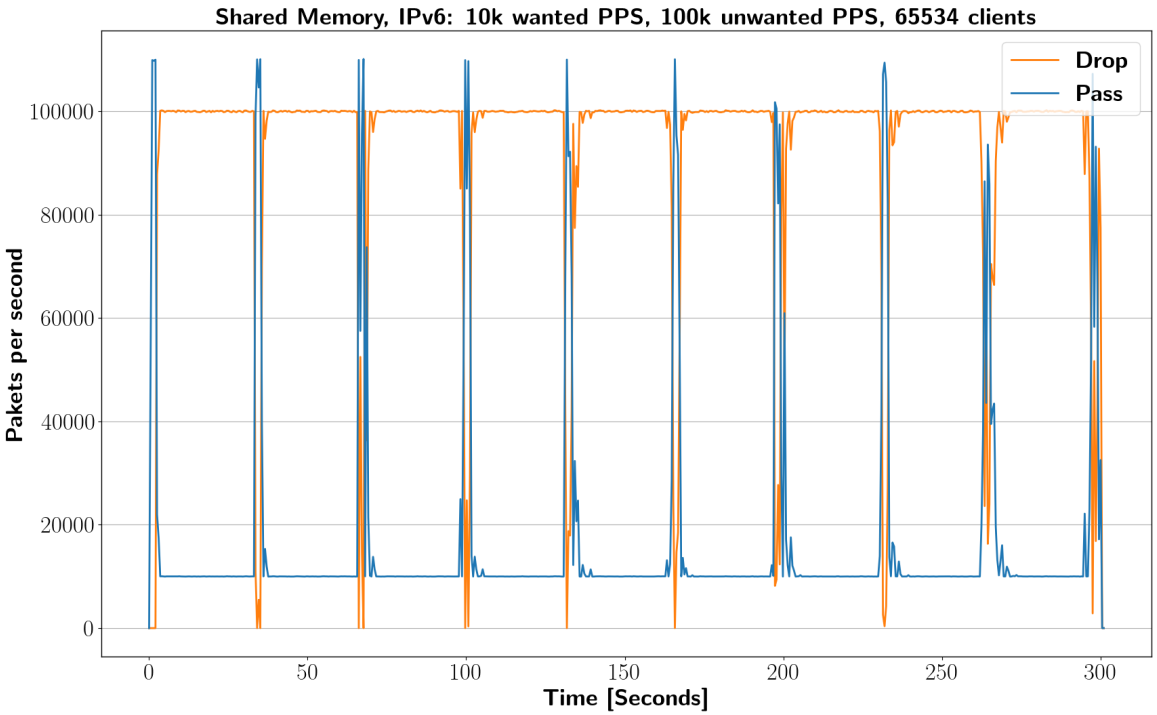
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
303	294.34	98.88	4.61	15.52

Figure 4.13: Some text



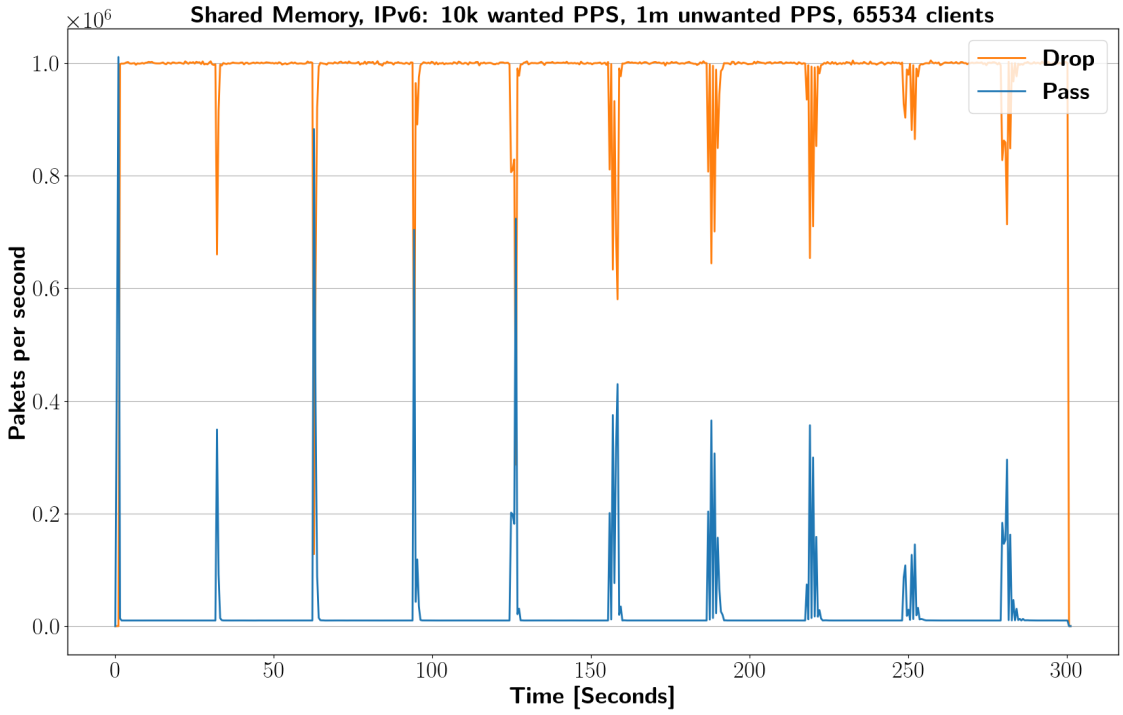
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
2991.48	2948.51	96.72	7.69	25.59

Figure 4.14: Some text



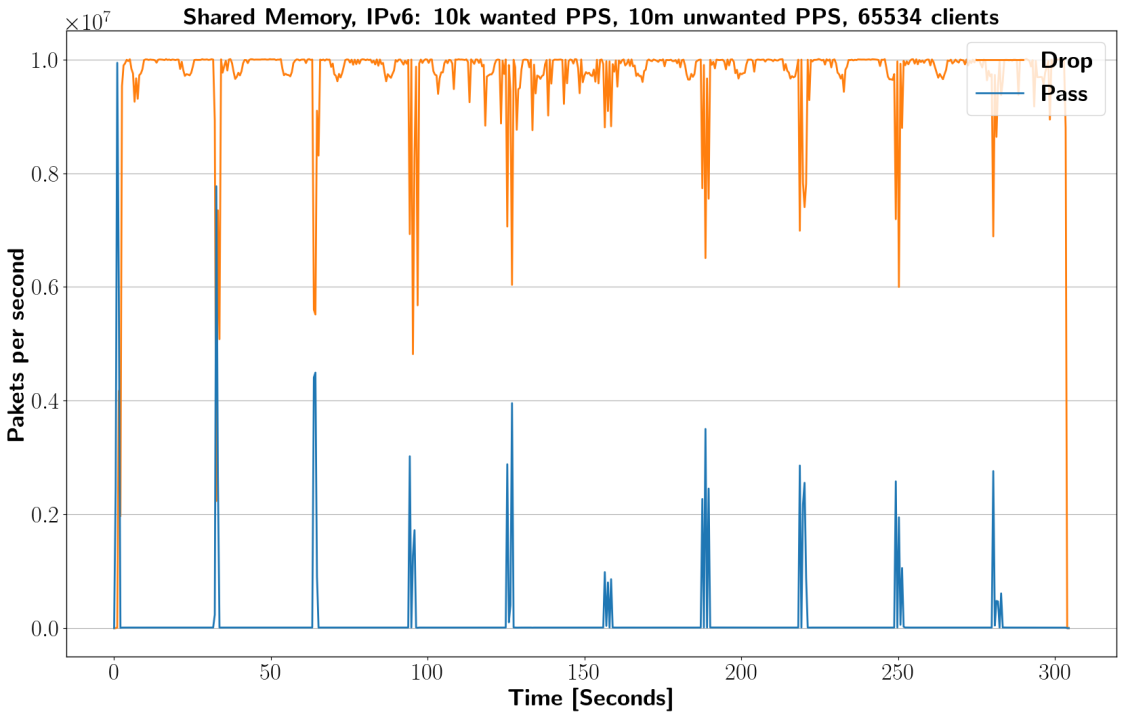
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
33	27.95	99.67	2.05	15.55

Figure 4.15: Some text



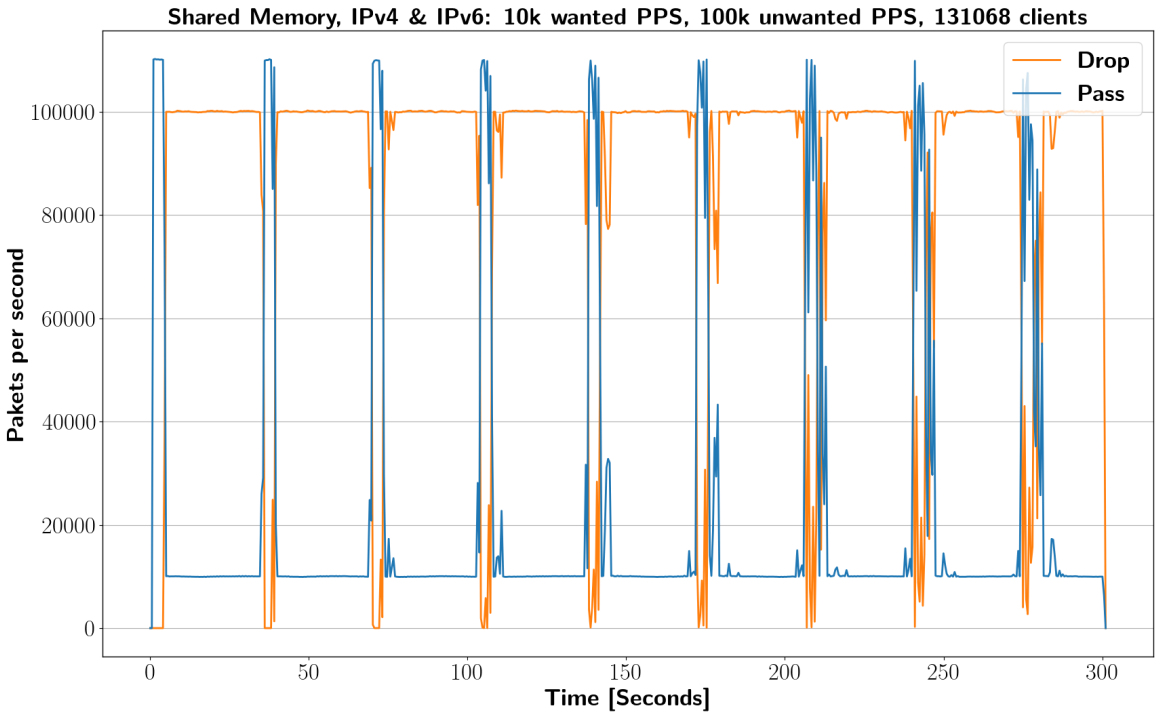
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
303	294.77	98.9	4.39	17.46

Figure 4.16: Some text



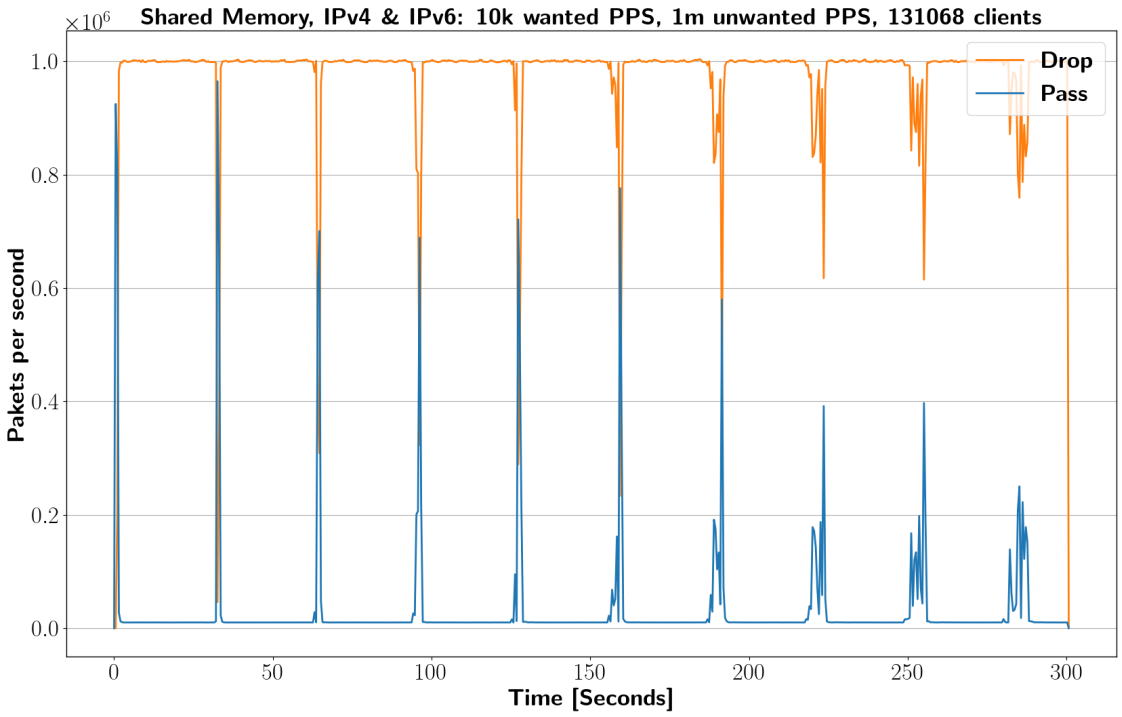
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
2991.93	2947.1	98.66	8.88	32.36

Figure 4.17: Some text



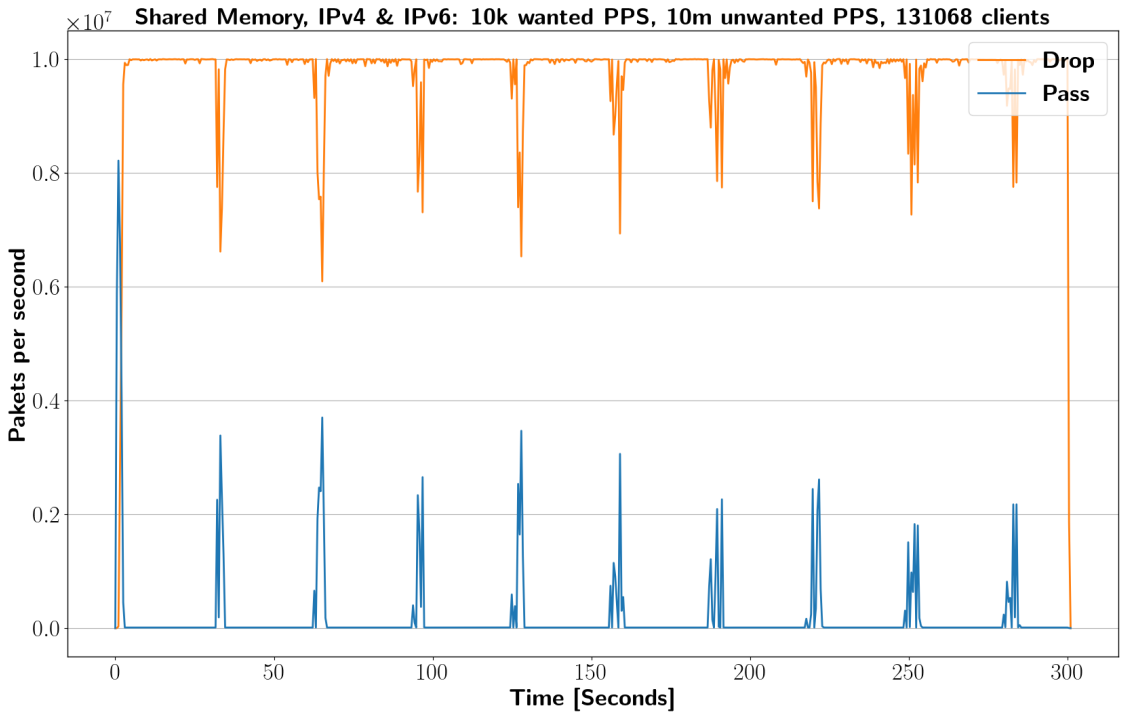
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
33	26.45	99.95	3.55	26.35

Figure 4.18: Some text



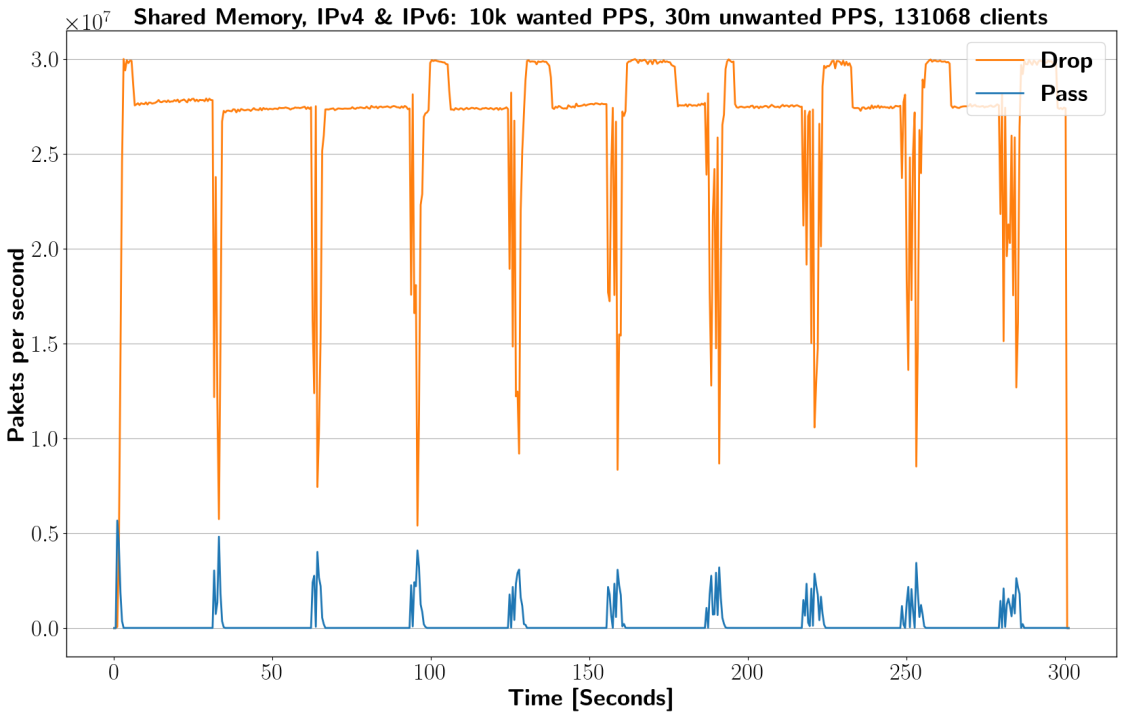
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
303	293.13	99	5.95	27.66

Figure 4.19: Some text



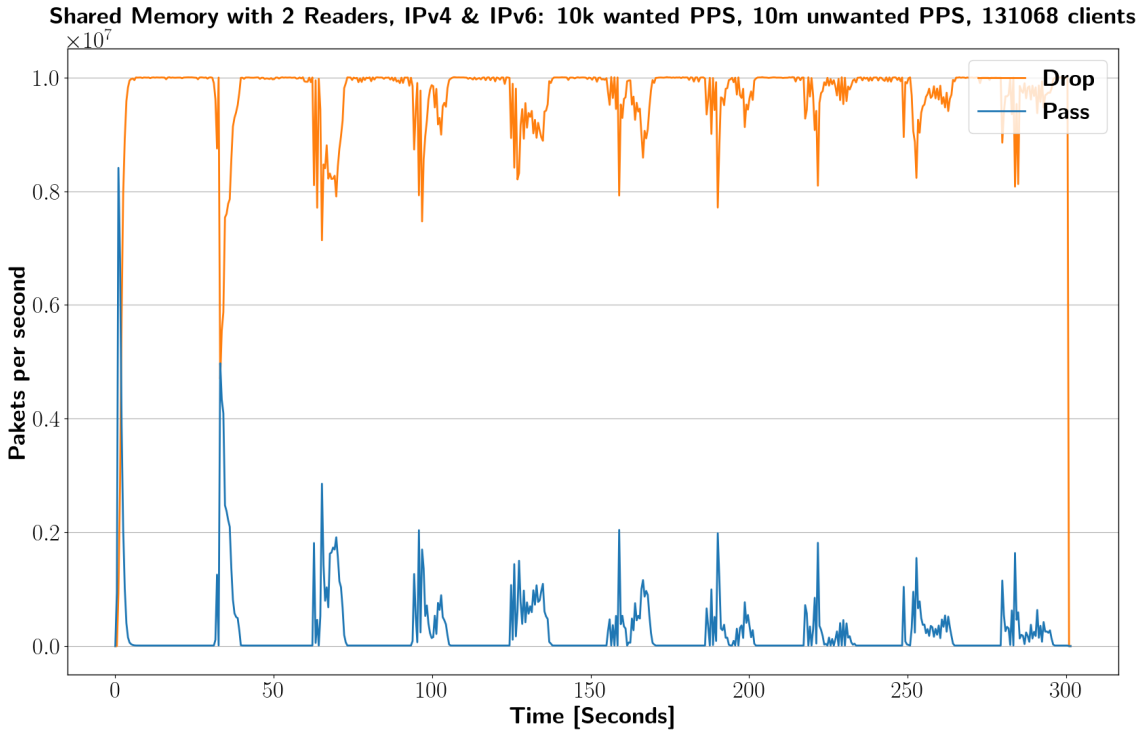
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
2992.44	2939.27	98.45	16.18	65.74

Figure 4.20: Some text



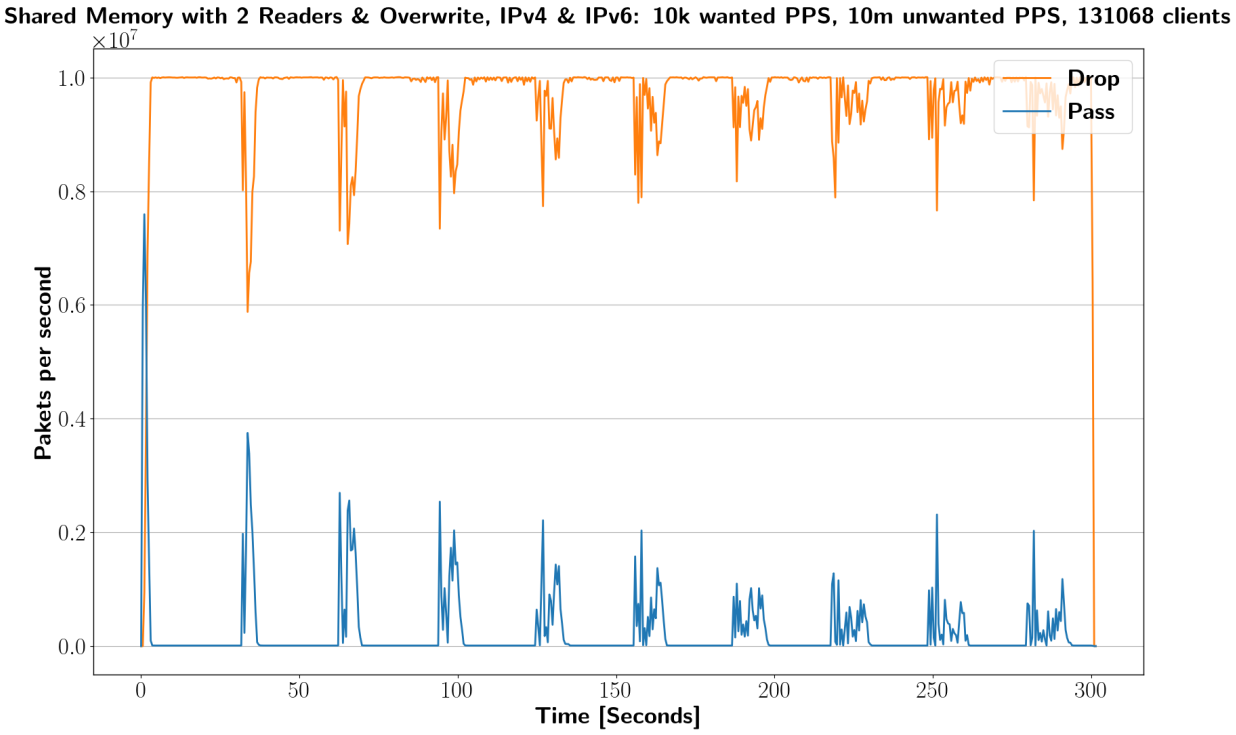
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
8095.28	8017.72	99.13	21	76.23

Figure 4.21: Some text



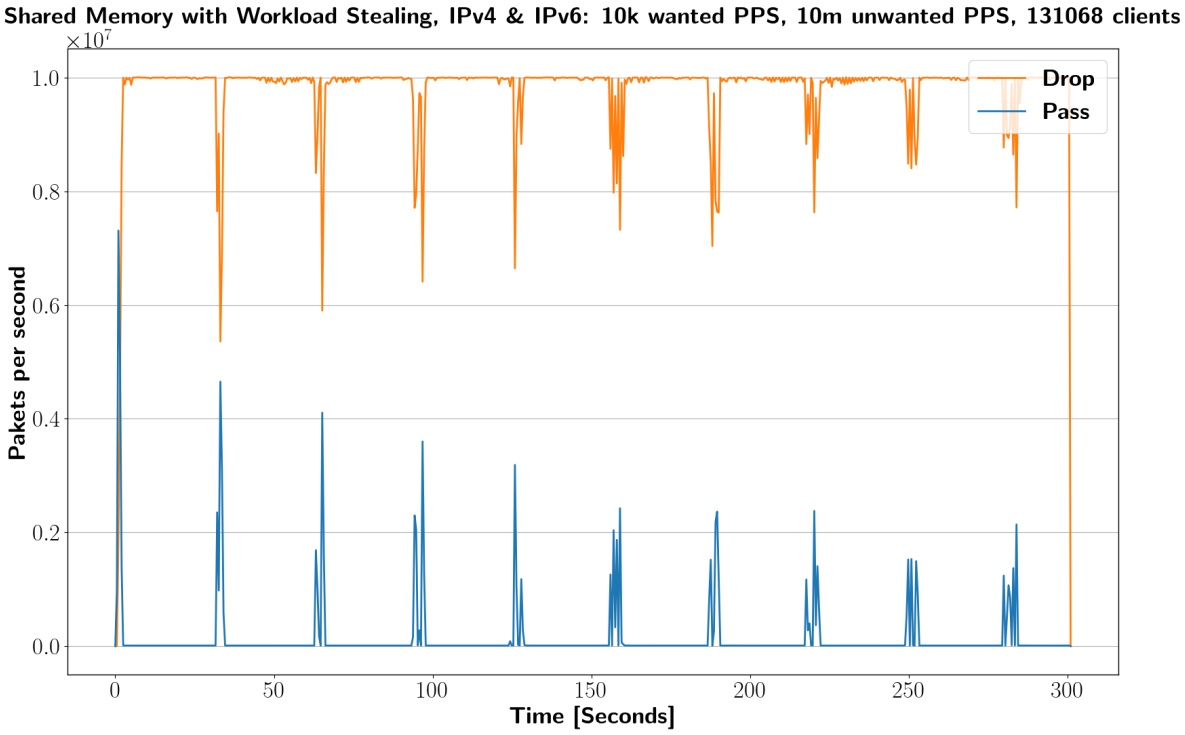
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
2989.66	2906.36	97.44	17.67	77.19

Figure 4.22: Some text



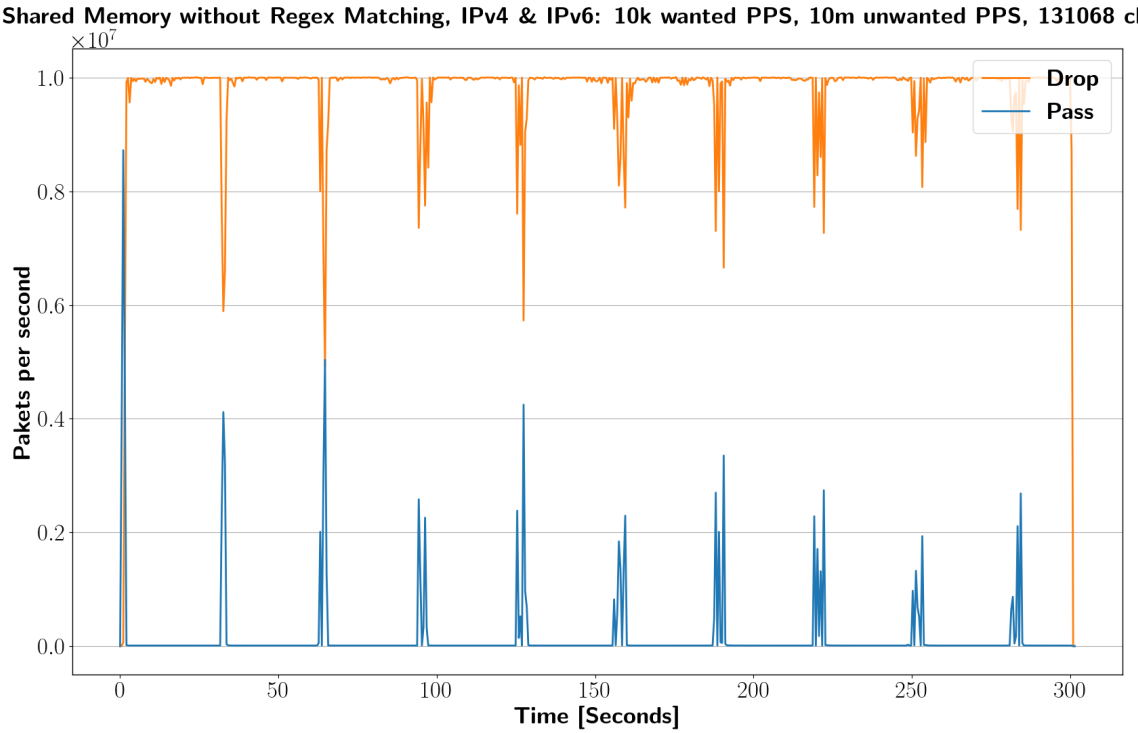
Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
2988.9	2914.39	97.73	17.66	79.13

Figure 4.23: Some text



Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
2991.23	2944.56	98.67	15.28	61

Figure 4.24: Some text



Total packets [10^6]	Packets dropped [10^6]	Relative drop [%]	Log messages [10^6]	CPU [seconds]
2989.56	2941.9	98.63	13.29	34.04

Figure 4.25: Some text

5 Conclusion

List of Figures

List of Tables

List of Algorithms

A Abbreviations

DUT	Device under Test
DoS	Denial of Service
eBPF	extended Berkeley Packet Filter
FIFO	First in First out
HIDS	Host-based Intrusion Detection System
IP	Internet Protocol
IPC	Inter-Process Communication
IPS	Intrusion Prevention System
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
IO	Input / Output
NIDS	Network-based Intrusion Detection System
OS	Operating System
PoC	Proof of Concept
PPS	Packets per Second
REGEX	Regular Expression
SIEM	Security Information and Event Management
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
XDP	eXpress Data Path

B Source Files

For the sake of not having to chop down a forest to print this thesis, no full source files will be appended. The source code is available in a git repository at: https://gitup.uni-potsdam.de/raatschen/bachelorarbeit. Access can be requested through me, or the second supervisor Max Schrötter.

Bibliography

[1] Fail2Ban. Official Fail2Ban Website. https://www.fail2ban.org, 2011. Last visited on: 14.04.2023.

[2] Giovanni Vigna and Christopher Kruegel. Host-based intrusion detection. In *Handbook of Information Security*, pages 701–713. California State University, 2006.

[3] James P Anderson. Computer security threat monitoring and surveillance. *Technical Report, James P. Anderson Company*, 1980.

[4] Dorothy E Denning. An intrusion-detection model. *IEEE Transactions on software engineering*, pages 222–232, 1987.

[5] Florian Mikolajczak. Implementation and Evaluation of an Intrusion Prevention System Leveraging eBPF on the Basis of Fail2Ban. https://www.cs.uni-potsdam.de/bs/teaching/docs/thesis/2022/mikolajczak.pdf, 2022. Master Thesis, Institute for Computer Science Operating Systems and Distributed Systems, University of Potsdam.

[6] W Richard Stevens. *UNIX network programming, Volume 2, Interprocess Communications*. Prentice Hall, NJ, USA, 1998.

[7] Linux man-pages project. Systen V IPC Manpage. https://man7.org/linux/man-pages/man7/sysvipc.7.html, 2020. Last visited on: 14.04.2023.

[8] Linux man-pages project. Posix Shared Memory Manpage. https://man7.org/linux/man-pages/man7/shm_overview.7.html, 2021. Last visited on: 14.04.2023.

[9] R Recio, B Metzler, P Culley, J Hilland, and D Garcia. Rfc 5040: A remote direct memory access protocol specification, 2007.

[10] Linux man-pages project. Pipe Manpage. https://www.man7.org/linux/man-pages/man7/pipe.7.html, 2021. Last visited on: 14.04.2023.

[11] W Richard Stevens. *UNIX network programming, Volume 2, Network Programming*. Prentice Hall, NJ, USA, 1998.

[12] Linux man-pages project. Unix Socket Manpage. https://man7.org/linux/man-pages/man7/unix.7.html, 2021. Last visited on: 14.04.2023.

[13] Linux man-pages project. POSIX Message Queue Manpage. https://man7.org/linux/man-pages/man7/mq_overview.7.html, 2020. Last visited on: 14.04.2023.

[14] ZeroMQ. Official ZeroMQ Website. https://zeromq.org/, 2022. Last visited on: 14.04.2023.

[15] Intel. Hyperscan. https://github.com/intel/hyperscan. high-performance regular expression matching library.

[16] Paul Raatschen. Thesis git repository. https://gitup.uni-potsdam.de/raatschen/bachelorarbeit, 2023.

[17] Linux man-pages project. Perf Manpage. https://www.man7.org/linux/man-pages/man1/perf.1.html, 2022. Last visited on: 14.04.2023.

[18] Linux man-pages project. Getline Manpage. https://www.man7.org/linux/man-pages/man3/getline.3.html, 2021.

[19] Andi Kleen. Spooky-c. https://github.com/andikleen/spooky-c. C version of Bob Jenkins' spooky hash.